Second Edition

C++ Programming Made Simple

Incorporating Standard C

Conor Sexton

MADE SIMPLE BOOKS

Made Simple
An imprint of Elsevier Science
Linacre House, Jordan Hill, Oxford OX2 8DP
225 Wildwood Avenue, Woburn MA 01801-2041

First published 2003

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 0 7506 5738 3

Typeset by Elle and P.K. McBride, Southampton
Icons designed by Sarah Ward © 1994
Printed and bound in Great Britain

Contents

Preface

C++ Programming Made Simple – Second Edition is intended as an introduction to programming in the C++ language as codified by the 1998 ISO C++ Standard. It is not a reference book and does not pretend to be in any way comprehensive. The intention is to provide an accessible starting-point to people who:

- have no programming experience
- have programmed in some other high-level language
- know C or an earlier version of C++ and need an update
- in any case need a working, practical, knowledge of C++.

This book owes a good deal to its two 1997 predecessors *C Programming Made Simple* and *C++ Programming Made Simple*. At the time I wrote these, it was still customary to present C++ as an extension of C. Because C++ was then relatively new, many people were in a position of knowing C and needing an 'upgrade' to C++. This situation has now changed. The distinction between C and C++ has blurred if not disappeared: in the ISO Standard, C++ incorporates C. It can no longer be assumed that people will have a knowledge of C before approaching C++. What is called for – and, I hope, provided by this book – is an integrated coverage of C++ including the parts of C that are not obsolete.

Achieving this has involved much more than simply merging the two previous texts. First, C++ had priority: where a C idiom is replaced by a newer C++ construct, the former is no longer covered. Second, a great deal has changed in C++ in the six years since the original two *Made Simples*. The language has been 'tweaked' in a thousand details. The C++ Library and the Standard Template Library in particular are largely new. A complete revision was necessary; I hope that it is evident that this is a new book and not just a re-hash of two old ones.

This book does not try to take a rigorous approach to C++. Coverage of the language aims to be adequate for practical needs, not complete. Many of the 'dark corners' of C++ (and some not-so-dark ones) are not covered. Even with this selectiveness, the book still comes out at over 300 pages. I estimate that a completely comprehensive coverage of modern C++, including the Standard Library, would weigh in at more than 1,500 pages. The objective, then, is to get you 'up and running' with useful C++ grounded in clear, practical, program examples.

The book has 14 chapters, which I don't list here: you can see them in the Table of Contents. I think of it as having four main parts:

- In time-honoured fashion, a lightning overview of C++ essentials (Chapter 1)
- The 'C-heavy' part of the book, with C++ syntax integrated as appropriate (Chapters 2 to 7)
- Traditional C++, involving classes and inheritance (Chapters 8 to 10)
- Modern C++, including templates and the Standard Library (Chapters 11 to 14)

At those points where a topic could grow beyond the scope of a *Made Simple* book, I acknowledge the fact, and make suggestions for further reading.

I have enjoyed the various aspects of writing this book: using some material from the previous *Made Simples*, eliminating obsolete aspects of C; updating the C++ syntax and adding completely new sections. It may be optimistic of me to ask you to enjoy reading it; I hope at least that you find it useful.

Conor Sexton
Dublin

1 A quick start with C++

Background to ISO C++ language

The C++ programming language is an *object-oriented* (OO) derivative of C. It is almost true to say that C is a subset of C++. In fact, every ISO C program written in the modern idiom (specifically, with new-style function headers) and avoiding certain C++ reserved words is also a C++ program, although it is not object-oriented.

This book owes a good deal to two of my previous publications in the *Made Simple* series, *C Programming Made Simple* (0-7506-3244-5) and *C++ Programming Made Simple* (0-7506-3243-7) both published by Butterworth-Heinemann in 1997. Since that time, and, particularly, since the September 1998 ratification of the ISO C++ Standard, C has been completely subsumed by C++. C as a language no longer exists in its own right; it is therefore no longer acceptable to take what was once the conventional approach and treat C and C++ separately. Accordingly, this book presents a fully-integrated treatment of the ISO C++ language incorporating C. Parts of the original C language and library are still valid but have been replaced by superior C++ facilities. Examples of such C constructs include void parameter lists and library functions including printf, malloc and others. This book does not deal with the (obsolescent) C mechanisms, but concentrates on the facilities provided by C++. Finally, from now on in this book, the simple term 'C++' should be taken to mean 'ISO C++'

It is the aim of this first chapter to get you up and running quickly with C++. The remaining chapters go into somewhat more depth on a variety of C++ constructs and programming techniques.

C++ is for technical computer programming and is suitable for development of 'techie' software like operating systems, graphical interfaces, communications drivers and database managers. In the modern Web context, C++ is one of the three or four languages of choice for implementation of applications in the so-called *middle tier*. These applications constitute what is often called the *business logic*: the body of application code resident on an intermediate system between the (browser-based) user and the (typically database) resource at the *back end*.

The main alternative to C++ is Java. Java is the medium in which the Java 2 Enterprise Edition (J2EE) web application architecture is implemented. Java has an advantage over C++ in not having to be compiled for every type of computer on which programs written in the language must run; it is therefore more easily *portable* (movable between different systems) than C++ and particularly suitable for web applications.

At the time of writing, there are two main web application architecture 'camps', J2EE – originating from Sun Microsystems Inc. – and Microsoft's C# (pronounced 'C-sharp') and .Net combination. J2EE exclusively uses Java; C#/.Net allows use of C++ (called 'Managed C++'), C# and Visual Basic. The predecessor to the C#/.Net architecture, the Component Object Model (COM), is still widely in use and employs C++ and Visual Basic as its two primary languages.

Although Java has an advantage of portability over C++ in the web applications context, C++ is still widely present on the web, very often written in the form of COM objects or CGI (Common Gateway Interface) programs. C++ also retains the advantages of performance and flexibility over Java. Line for line, because Java is not compiled into an optimised executable form, C++ is likely to be faster in execution on a given system. In addition, C++ retains constructs (such as pointers, eliminated by Java) that allow it full access to all operating system and machine facilities, with corresponding flexibility and power.

C++ was originally developed in the early 1980s at AT&T Bell Laboratories by Dr Bjarne Stroustrup. In the almost 20 intervening years, there has been a myriad of twists and turns to the development and standardisation of the language. Mostly, these are no longer important. It's enough to refer to the standardisation process, which was started by the American National Standards Institute (ANSI) in 1990 when it formed the standardisation committee X3J16. About the same time, the International Organization for Standardization (ISO) formed its committee ISO-WG-21, also for the purpose of standardising C++ on a worldwide basis. The efforts of the two committees were made joint from 1990 and it was at the time expected that a ratified ANSI/ISO C++ standard would be approved by 1994. In the event, there were significant additions to the scope of the work involved – most notably the addition of the C++ Standard Library including the Standard Template Library (STL) – and the Final Draft International Standard (FDIS) was not published until late 1997. The International Standard (IS) was ratified in September 1998. Its ISO title is *Information Technology – Programming Languages – C++*, with associated document number ISO/IEC 14882-1998. Though originated by ANSI, the standard is an ISO one; the documented one is distributed by national standards bodies subordinated to ISO (ANSI in the case of the United States).

ISO C++ (also called 'Standard C++') is now the single unified definition of the C++ language, and it is becoming increasingly difficult to find books and compilers that do not at least claim to conform to the Standard. The first edition of this book (*C++ Programming Made Simple*, 1997) is not ISO-Standard compliant, but is close to being so. If you know to make a few small but significant changes to the structure and syntax of programs, that book still serves as a viable presentation of the C++ language. This edition presents and explains the necessary changes, as well as a subset of the extensions (mostly in the area of the Standard Library) that are mandated by the Standard.

Some of the major characteristics of C++ are these:

- C++ provides a powerful, flexible and expressive *procedural* language (alongside an object-oriented or class-based component) grounded in the earlier C language.
- C++ implements *objects*, defined as classes, which incorporate data definitions along with declarations and definitions of functions that operate on that

data. This *encapsulation* of data and functions in a single object is the central innovation of C++.

- *Instances* of classes may automatically be initialised and discarded using *constructors* and *destructors*. This eliminates program initialisation errors.
- The way in which C++ classes are defined enforces *data hiding*; data defined in a class is by default only available to the *member functions* of that class. External, or *client*, code that uses a class cannot tamper with the internal implementation of the class but is restricted to accessing the class by calling its member functions.
- C++ allows *overloading* of operators and functions. More than one definition of a function may be made having the same name, with the compiler identifying the appropriate definition for a given function call. Ordinary operators such as ++ and += can also be overloaded with additional meanings.
- C++ allows the characteristics of the class type — data and functions — to be inherited by *subclasses*, also called *derived classes*, which may in turn add further data and function definitions. This encourages reuse of existing code written in the form of shareable class libraries and consequent savings in the cost of the software development process. *Multiple inheritance* allows for derived classes to inherit characteristics from more than one base class.
- C++ allows classes to define *virtual functions*: more than one definition of a function, with the decision as to which one is selected being resolved at program run-time. This is *polymorphism*, with the run-time selection among function definitions being referred to as *late binding* or *dynamic binding*.
- *Template* classes can be defined which allow different instances of the same class to be used with data of different types but with unchanged code. This further promotes code reuse.
- ISO C++ introduces the standardised C++ Library, which includes the Standard Template Library (STL). The Standard Library incorporates and improves the old (see the first edition of this book) Stream I/O library and adds many utilities and other features. The STL provides programmer-friendly implementations of many common data structures — including list, stack, queue, string — along with the operations necessary to manipulate them. Thus, many of the 'cool' programming techniques familiar to advanced C programmers of the 1980s and early 1990s are now packaged and made invisible for you, much in the same way that the operation of a car's engine is hidden from the average owner by the bonnet (or hood, depending on where you live)!

C++ facilities for object-oriented programming are characterised by classes, inheritance and virtual functions. These facilities make C++ particularly suitable for writing software to handle a multitude of related objects. A typical use of C++ is in implementing graphical user interfaces (GUIs), where many different but related objects are represented on a screen and are allowed to interact. Using the object oriented approach, C++ stores these objects in class hierarchies and, by means of virtual functions, provides a generic interface to those objects (e.g. draw object), which saves the programmer from having to know the detail of how the objects are manipulated. This makes it easier for the programmer to develop and maintain code, as well as rendering less likely the introduction of bugs into existing code. C++ and other object-oriented languages are also, as we have seen, central to the modern component-based architectures such as .Net and J2EE, in which reuse of proven and tested objects improves prospects for reliability of applications of ever-increasing complexity.

That's enough overview stuff. Let's write our first program!

The do-nothing program

The minimal C++ program is this

```
main(){}
```

This is a complete C++ program. Every C++ program must consist of one or more functions. The code shown above is a program consisting exclusively of a main function. Every C++ program must have one (and only one) main function. When it is executed, the program does nothing.

A more strictly correct C++ form of the do-nothing program is this:

```
#include <iostream>
int main(){return 0;}
```

The whole program shown is stored in a file called donowt.cpp. The .cpp part is necessary, meaning that the file contains a C++ program; donowt is at your discretion. iostream is a *standard header file* that contains useful declarations for compilation and execution of the program that follows. If you are aware of the syntax of pre-ISO C++, you'll know that the name of the header file was iostream.h; this form is still usable.

iostream is an alternative to but does not replace the C standard header file cstdio. Again, in pre-ISO C++, this was called stdio.h; the .h has now been removed at the behest of the C++ standardisation committee and the leading 'c' added to make clear the header file's C Library lineage. iostream declares C++ library functions and facilities (see Chapters 12 and 13); cstdio does the same for Standard C Library functions such as printf. The int preceding main is the function's *return type*. It specifies that the program returns a value (in this case, zero) to the operating system when it is run. The function's parentheses are empty: the function cannot accept any parameters. When the program finishes executing (it does nothing), the return statement returns execution control to the operating system.

Here's the most-correct (by ISO criteria) version of donowt:

```
#include <iostream>
using namespace std;
int main(){}
```

The standard, or std, *namespace* is introduced. This has no practical effect as yet, but its purpose is to allow objects of the same name to be used in different contexts without a name clash. You might want to use the standard output object cout to mean two different things, so the first version would belong to the standard namespace, while the second would belong to another namespace. More of this later; for now (and for most of the book) we'll confine ourselves to the standard namespace.

Notice also that the return statement is gone. ISO C++ no longer requires this. You may include the line but, if you don't, the C++ system inserts an implicit one for you and ensures that control is correctly returned to the operating system.

Here is a rather complex version of the do-nothing C++ program donowt.cpp. It uses a trivial C++ class to produce no output

```
//   donowt.cpp - program using a simple C++
//   class to display nothing

#include <iostream>
using namespace std;

class nodisp
{
private:
public:
     void output()
     {
          return;
     }
};

int main()
{
     nodisp screen;

     screen.output();
}
```

This time the program declares a C++ class, nodisp, of which the only member is a function output. In the main function, we define an instance of the class

    nodisp screen;

The function nodisp::output() (the member function output of the class nodisp) when executed from main with the line screen.output(), simply returns control to the following statement. As this is the end of main, donowt.cpp stops without making any output.

Brackets and punctuation

A note on the different kinds of brackets: the names of standard header files are enclosed in angle brackets <>, function argument lists after the function name, in parentheses (), and code blocks in curly braces {}. Statements such as return must be terminated with a semicolon. C++ programs are free-form – you can write the code in any format, jumbled up on one line of text if you want. The structured layout shown in this book is not strictly necessary but is a good idea for readability and avoidance of error

Building and running a C program

The filename suffix for C++ programs is usually .cpp on PCs. On computers running the UNIX operating system, the C++ source code filename may end with any of several suffixes, including .c, .C, .cxx and .cpp. This book uses exclusively the suffix .cpp.

The donowt.cpp program, in any of the forms shown above, must first be converted by compiler and linker programs into executable code. For this book, I'm assuming use of the Borland C++ Builder 5 development suite. This has a vast range of supporting facilities for development of GUI and component-based objects in C++. It also has a very good command-line compiler, which is what I use in this book. If you're using a PC with the Borland C++ Builder 5 compiler and linker, you can compile donowt.cpp using this command line:

```
bcc32 donowt.cpp
```

This produces an output file called donowt.exe, which you can run at the command line, admiring the spectacular lack of results that ensues.

For some Microsoft C++ compilers, you can use 'c-ell':

```
cl donowt.cpp
```

More usually, you use the *integrated development environment* (IDE) provided by the Microsoft Visual C++ 6.0 or .Net environments.

If you're using a UNIX system, you can compile and *load* (UNIX-speak for link) the program using a number of different command-line formats, depending on the UNIX variant. I can't specify the precise command-line input for your UNIX system, so I present a few possibilities below (note that UNIX distinguishes between upper- and lower-case characters entered at the command line):

```
CC donowt.C        // UNIX System V
g++ donowt.cpp     // Linux
c++ donowt.C       // Also Linux; .cxx and .c suffixes OK too
```

Whichever command line is correct for your system (you'll have to experiment a bit), the resulting executable program is in a file called a.out (for *assembler output*, believe it or not).

Some programs can be built (compiled and linked) at the command line in the ways shown. Programs that make GUI displays, as well as programs for the modern component environments such as Microsoft's .Net, cannot in practical terms be built using the command line. It's more likely that you will use an IDE provided by Microsoft, Borland, IBM or another supplier. The IDE uses a menu-driven interface that is better for managing programs of significant size.

It's not the subject of this book to tell you how to use the IDEs of any software supplier. I assume that the information you now have will enable you to build at least simple C++ programs, and we move forward now to writing programs that actually do something.

Here's one, called message1 cpp

```
//   message1 cpp - program to display a greeting

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello C++ World\n";
}
```

The double-slash notation is a comment. all characters following the double slash // on the same line are ignored The /*   */ notation is also used in C++ but, for short comments. // is preferred.

The header file iostream contains class and function declarations that are #included by the *preprocessor* in the source code file and are necessary for C++ Library facilities to be used These facilities include cout of the class type ostream ostream is declared in the iostream header file.

cout is an object representing the *standard output stream* The characters to the right of the << operator are sent to cout which causes them to be displayed on the user's terminal screen assuming that is the standard output device The << operator is in fact the bitwise left-shift operator *overloaded* by the C++ system to mean insert on a stream The C++ Stream I/O system is explained in Chapter 13

You should try entering this program at your computer and building it As an exercise, make message cpp display two lines.

```
Ask not what your country can do for you
Ask rather what you can do for your country
```

If you were to omit the line

```
using namespace std.
```

you would get a compilation error complaining about cout being an "undefined symbol" This is because cout needs to be specified as being part of some namespace This can be specified explicitly as

```
std: cout << "Hello C++ World\n"
```

with the same result as if using namespace   had been retained

Here's a class-based version of the message program message2 cpp. that produces exactly the same output Hello C++ World as the simpler form of the program shown above

```
//   message2.cpp - program using a simple C++
//   class to display a greeting

#include <iostream>
using namespace std;

class message
{
private:
public:
     void greeting()
     {
          cout << "Hello C++ World\n";
     }
};

int main()
{
     message user;

     user.greeting();
}
```

You'll see much more about classes throughout this book and, in particular, in Chapter 8. For now, I'll briefly describe the message class and its contents. Everything within the enclosing curly braces following message is a member of the class message. All the members of message are declared public: they are generally accessible. message only has one public member, a function called greeting which has no return type or argument list.

In the function main, we define an instance of the message class, called user. The greeting function is called and the Hello C++ World message displayed by the function call:

```
user.greeting();
```

Use of a class in this case is overkill, but from it you should be able to understand simple characteristics of the class construct.

Enough C++ to get up and running

While **message2.cpp** does produce a visible result, it's not very useful. To produce more functional C++ programs, you must know a minimum set of the basic building blocks of the C++ language. This section presents these building blocks, under a number of headings:

- Variables
- Operators
- Expressions and statements
- Functions
- Branching
- Looping
- Arrays
- Classes
- Constructors and destructors
- Overloading
- Inheritance

Variables

Variables in C++ are data objects that may change in value. A variable is given a name by means of a definition, which allocates storage space for the data and associates the storage location with the variable name.

The C++ language defines five fundamental representations of data:

boolean

integer

character

floating-point

double floating-point

Each of these is associated with a special *type specifier*:

bool	specifies a true/false value
int	specifies an integer variable
char	specifies a character variable
float	specifies a fractional-number variable
double	specifies a fractional-number variable with more decimal places

Any of the type specifiers may be qualified with the type qualifier **const**, which specifies that the variable must not be changed after it is initialised.

A data definition is of the following general form:

```
<type-specifier> <name>;
```

A variable name is also called an *identifier*. The following are some examples of simple data definitions in C++:

```
int       apples,            // integer variable
char      c                  // character value eg. 'b'
float     balance            // bank balance
const     double x = 5;      // high-precision variable
                             // value fixed when set
bool      cplusplus = TRUE
```

Operators

C++ has a full set of arithmetic, relational and logical operators. The binary arithmetic operators in C++ are

+	addition	-	subtraction
*	multiplication	/	division
%	modulus		

There is no operator for exponentiation in line with general C++ practice, this is implemented as a special function in an external library

Both + and - may be used as unary operators, as in the cases of -5 and +8. There is no difference between +8 and 8.

The modulus operator, %, provides a useful remainder facility

```
17%4 // gives 1, the remainder after division
```

The assignment operator, =, assigns a value to a memory location associated with a variable name. For example

```
a = 7,
pi = 3 1415927,
```

Relational operators in C++ are

<	less than	>	greater than
>=	greater than or equal to	<=	less than or equal to
!=	not equal		
==	test for equality		

Care is needed in use of the equality test ==. A beginning programmer will at least once make the mistake of using a single = as an equality test, experienced programmers do it all the time!

Writing

```
x = 5.
```

assigns the value 5 to the memory location associated with the name x

The statement

```
x == 5;
```

on the other hand, tests the value at the memory location associated with the name x for equality with 5. Confusion here can result in serious program logic errors. It is a good idea, with the editor, to check all usages in the source code of = and == manually. The compiler will not catch these mistakes for you.

True and false

Logical operators provided by C++ are:

&&	AND
||	OR
!	NOT (unary negation operator)

If two variables are defined and initialised like this:

```
int x = 4;
int y = 5;
```

then

```
(x == 4) && (y == 5)        is TRUE
(x == 4) || (y == 3)        is TRUE
!x                          is FALSE
```

In C++, any non-zero variable is inherently TRUE; its negation is therefore FALSE. The quantities TRUE and FALSE are not themselves part of the C++ language. You can define them with the preprocessor

```
#define TRUE 1
#define FALSE 0
```

or as const-qualified declarations:

```
const int TRUE = 1;
const int FALSE = 0;
```

By convention in C++, truth is defined as non-zero and falsehood as zero. This, unfortunately, is the opposite of the interpretation adopted by operating systems including UNIX. Thus, while a C++ program will use zero internally to represent a failure of some kind, it will probably, when it terminates, return zero to the operating system to indicate success.

C++ also supplies the bool type and the associated true and false keywords, which can be used instead of the more traditional preprocessor form.

```
//   boolean.cpp - program to test bool, 'true' and 'false'

#include <iostream>
using namespace std;

int main()
{
     int x = 4;
     int y = 5;

     if (((x == 4) && (y == 5)) == true)
          cout << "Both\n";
     if (((x == 4) || (y == 3)) == true)
          cout << "Just one\n";
     if ((!x) == false)
          cout << "Not-X is false\n";
}
```

The displayed output of this program is:

```
Both
Just one
Not-X is false
```

Expressions and statements

An expression is any valid combination of function names, variables, constants, operators and subexpressions. A simple statement is an expression terminated by a semicolon.

The following are all expressions:

```
a = 5
cout << "Hello World\n"
a = b + c
a = b + (c * d)
```

Every expression has a type, depending on the types of its constituents, and a boolean value. The expression:

```
a = b + c;
```

assigns to a the sum of the values of variables b and c. Expressions in C++ can be complex. Here is a slightly less simple one:

```
a = b + c * d
```

In this case, the order of arithmetic evaluation is important:

```
a = b + (c * d)
```

is not the same as

```
a = (b + c) * d
```

because the *precedence* of the operators is different. We can summarise the order of precedence of common C++ operators as follows:

()	Sub-expressions surrounded with parentheses (high precedence)
! -	The unary negation operator and unary minus
* / %	The arithmetic operators
+ -	The plus and minus binary arithmetic operators
< <= > >=	The relational operators
!= ==	The equality operators
&& ||	The logical operators (low precedence)

Statements may optionally be grouped inside pairs of curly braces {}. One or more statements so grouped form a *compound statement*:

```
{
    cout << "Two statements...\n";
    cout << "that are logically one\n"
}
```

That a compound statement is a single logical entity is illustrated by the conditional statement:

```
 if (s == 2)
{
    cout << "Two statements...";
    cout << " that are logically one";
}
```

If the variable s has the value 2, both output lines are executed. Where the two statements are simple and not compound:

```
if (s == 2)
cout << "Two statements... ";
cout << " that are logically distinct")
```

the second output statement is executed even if s is not equal to 2.

Functions

A function is a body of C++ code executed from another part of the program by means of a function call. Functions usually contain code to perform a specific action. Instead of duplicating that code at every point in the program where the action is required, the programmer writes calls to the function, where the single definition of the code resides. Every C++ program is a collection of functions and declarations.

main, as we've seen, is a special function: it must be present in every C++ program. When the program is run, the operating system uses main as the *entry point* to the program. main in turn usually contains calls to an arbitrary number of programmer-defined functions.

The following is a simple general form for all functions.

```
<returntype> <functionname>(<arglist>)
{
    <statements>
}
```

Here is a C++ program, twofunc.cpp containing two functions.

```
#include <iostream>
using namespace std;
void myfunc();      // 'myfunc' declaration
int main()
{
    cout << "Main function" << endl;
    myfunc();
}

void myfunc()
{
    cout << "Myfunc" << endl;
}
```

In this program, main contains two statements: first the cout we have already seen, followed by the second, a call to the function myfunc, which contains a further, slightly different, cout statement.

When it is run, this program displays the lines of text:

```
Main function
Myfunc
```

on the standard output device (usually the screen display).

The statement

```
myfunc();
```

is the call from main to the function myfunc.

On execution, control is passed to myfunc from main. When the single statement in myfunc has been executed, control is returned to the first statement in main after the function call. In this case, there is no such statement and the whole program immediately stops execution.

The function myfunc is expressed in three parts: the *declaration* (also called a function *prototype*):

```
void myfunc();
```

which announces to the compiler the existence of myfunc, the call

```
myfunc();
```

and the *definition* of the function itself:

```
void myfunc(void)
{
    cout << "Myfunc" << endl;
}
```

Note that the function call is a statement and must be terminated with a semi-colon. The prototype is not a statement but is distinguished from the header of the called function by a terminating semi-colon. The header must not be appended with a semi-colon. Every C++ function must be fully described in three parts using a declaration, call and definition.

Branching

You can use the if statement to allow decisions and consequent changes in the flow of control to be made by the program logic. The following is the general form of if:

```
if (<expression>)
    <statement1>
else
    <statement2>
```

The else part is optional: an if statement with one or more subject statements and no alternative provided by else is legal. For example:

```
if (nobufs < MAXBUF)
    nobufs = nobufs + 1;
```

Here, if the number of buffers used is less than the allowed maximum, the counter of used buffers is incremented by one. Two or more statements may be made subject to an if by use of a compound statement:

```
if (day == 1)
{
    cout << "Monday" << endl;
    week = week + 1;
}
if (day == 2)
{
    cout << "Tuesday" << endl;
    run_sales_report();
}
```

else should be used where the program logic suggests it.

```
if (day == 1)
{
    cout << "Monday" << endl;
    week = week + 1;
}
else
if (day == 2)
{
    cout << "Tuesday" << endl;
    run_sales_report();
}
```

Use of **else** here stops execution of the Tuesday code if the value of **day** is 1. **endl** in all the above cases is a C++ *manipulator* that has the effect of appending a newline to the text just displayed: the cursor moves to the next line as a result.

It's possible to nest if statements:

```
if (month == 2)
    if (day == 29)
        cout << "Leap Year!!" << endl;
    else
        cout << "February" << endl;
```

Nesting of ifs can be performed to arbitrary depth and complexity while the whole construct remains syntactically a single statement.

Looping

Where the if statement allows a branch in the program flow of control, the for, while and do statements allow repeated execution of code in loops.

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    x = 1;
    while (x < 100)
    {
        cout << "Number " << x << endl;
        x = x + 1;
    }
}
```

This program displays all the numbers from 1 to 99 inclusive.

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    for (x = 1; x < 100; x = x + 1)
        cout << "Number " << x << endl;
}
```

This program does exactly the same. The for statement is often used when the condition limits (in this case 1 and 100) are known in advance. The general form of the for statement is this:

```
    for (<expr1>;<expr2>;<expr3>)
        <statement>
```

Any of the expressions may be omitted, but the two semicolons must be included. For example, the statement:

```
    for ( ; ; );
```

results in an infinite loop.

The do statement is a special case of while. It is generally used where is it is required to execute the loop statements at least once.

```
    do
    {
        c = getchar();
        if (c == EOF)
            cout << "End of text" << endl;
        else
            /* do something with c */
    } while (c != EOF);
```

The *symbolic constant* EOF is defined in cstdio as the numeric value -1. The keystroke sequence required to generate this value is system-dependent. On UNIX systems, EOF is generated by *Ctrl-D*, on PCs by *Ctrl-Z*. Use of do instead of while is relatively rare: perhaps 5% of all cases.

The following example illustrates use of the C library functions putchar and getchar, as well as if and one of the iterative statements.

Notice the getchar function call embedded in the while condition expression. This is legal and also considered good practice in concise programming.

```
    /*
```

```
 *  Program 'copyio.cpp': copy standard input to
 *  standard output stripping out newlines
 */
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int c;

    while ((c = getchar()) != EOF)
    {
        if (c != '\n')
            putchar(c);
    }
}
```

Arrays

An array is an aggregate data object consisting of one or more data elements all of the same type. Any data object may be stored in an array. You can define an array of ten integer variables like this:

```
    int num[10];
```

The value within the square brackets, [], is known as a *subscript*. In the case above, ten contiguous (side-by-side) memory locations for integer values are allocated by the compiler. In this case, the subscript range is from zero to 9. When using a variable as a subscript you should take care to count from zero and stop one short of the subscript value. Failure to do this will result in unpleasant program errors. The following is a simple example of use of arrays:

```
/*
 *  'array.cpp': fill integer array with zeros, fill
 *  character array with blanks
 */

#include <iostream>
using namespace std;

int main()
{
    int   n[20];
    char    c[20];
    int   i;
```

```
        for (i = 0; i < 20; i = i + 1)
        {
            n[i] = 0;
            c[i] = ' ';
        }
}
```

Notice that i starts the iteration with value zero and finishes at 19. If it were incremented to 20, a memory location outside the bounds of the array would be accessed. No array bound checking is done by the C++ compiler or run-time system. To implement such checking, you have to implement the [] enclosing the array bounds as an *overloaded operator* (see later in this chapter and in Chapter 9 *Class services*).

A traditional *string* is a character array terminated by the null character '\0', also known as *binary zero*. (The C++ Library introduces the string class (see Chapters 2 and 12) which encapsulates in a standard class much of the functionality possible with traditional strings. Traditional strings are widely known as *C-strings*, reflecting the language in which they originated.) The C standard library (function declarations in the header files cstdio, cstring and cstdlib) contains many functions that perform operations on C-strings. Here are three:

```
gets(<string>);      // Read a string into an array
atoi(<string>);      // Convert ASCII to integer
atof(<string>);      // Convert ASCII to float
```

Using the following definitions:

```
char          instring[20]
int           binval
double        floatval
```

the statement

```
gets(instring);
```

reads from the standard input device a string of maximum length 20 characters, including the null terminator '\0'. There is nothing to stop the entry of data greater than 20 characters long; if there are more than 20 characters, the extra characters are written into whatever memory follows, perhaps causing this or another program to malfunction.

The terminated character array instring may then be converted into its integer numeric equivalent value using the library function atoi:

```
binval = atoi(instring);
```

instring may be converted into its double floating-point numeric equivalent value using the library function atof:

```
floatval = atof(instring)
```

Classes

C++ provides language support for the object-oriented programming (OOP) approach. A class consists of a number of *members*, which can be either variables or functions. You can use a class to describe a real-world object such as, for example, an insurance policy.

In the insurance business, a policy records information including (at least) the policy-holder's name and address, the policy number, the value of the entity insured and the premium to be paid for the insurance. At least four operations are possible on this information. You can open or close a policy, you can pay the premium to renew the policy, and you can make a claim. Using a C++ class, you could record the data and operations like this:

```
class policy
{
private:
    char name[30];
    char address[50];
    char polno[8];
    double ins_value;
    double premium;
public:
    void pol_open();
    void pol_close();
    void renew();
    bool claim(double);
};
```

This is a class declaration; in making it, you have informed the C++ compiler that the class exists, that instances of it will be of the format set down and that you may later want to make instances of the class. When an instance is created, 30 character spaces are reserved for the policy name, 50 for the address and 8 for the policy number. Items that may be fractional numerics are defined as variables of type double. Four functions are declared, one for each of the operations earlier specified.

You can define an instance of the class – often also referred to as a *class object* or *class variable* – like this:

```
policy myPolicy;
```

When coding the policy class, you would most likely store its declaration – shown above – in a header file (say, classes.h) #included in your program. You must then define – write the code of – each of the class's member functions in a program file (say, progfile.cpp). Here's how you might define the member function claim:

```
bool policy::claim(double amount)
{
    // check amount of claim is OK
    // pay claim to policy holder
```

```
        return(true);
    }
```

The actual code of the function is unimportant, which is why it's given as comments. What is relevant here is the form of the function header:

```
    bool policy::claim(double amount)
```

The *scope resolution operator* :: indicates that the function claim is a member function of the policy class. claim also returns a value of the boolean type bool (only true and false values are allowable) and takes a single parameter of type double.

When you've declared your class and defined the code of all its member functions, the full definition of the class is complete. With an instance of the class such as myPolicy, you can now make the claim:

```
    myPolicy.claim(1000000.0);
```

There are two parts to the class policy: private and public. The private keyword means that the class members declared following it are only accessible to member functions of the class policy — pol_open, pol_close, renew and claim. The public keyword means that any other function may make a call to any of these four functions.

The data hiding that is enforced by the private part of the class means that you can't access the private member functions from code other than that defined in the member functions of the policy class. All that is available to external (or client) code is the class's *function call interface*; the internal implementation of the class remains a *black box*.

This mechanism results in the production of highly modular code that you and other programmers can use without having to know anything about the code other than how to call it.

The general class policy can be refined using derived classes that take on the characteristics of policy and add new ones. For example, the class motor might add a reserve (the unpaid first part of a claim) and a no-claims-bonus function. And the class life might add a term or a fixed sum assured. This is an intuitive example of class inheritance.

Constructors and destructors

In the policy class example, you have to remember to open and close the policy when necessary. You do this by using an instance of the class to call the pol_open and pol_close functions. A common source of errors in all programs is when initialisation such as this is omitted.

In C++, automatic initialisation and discarding are done using constructors and destructors. A constructor is a member function of a class which initialises a variable of that class. The constructor function name is always the same as the class name. A destructor is a member function of a class which performs housekeeping operations before the class instance is itself destroyed. The destructor function name is the same as the class name and is prefixed with a tilde ~.

The constructor function is called as part of the definition of the class instance; the destructor is called not explicitly but automatically when the variable goes out of scope (is discarded).

Here is the policy class reworked to use constructors and destructors:

```
class policy
{
private:
    char name[30];
    char address[50];
    char polno[8];
    double ins_value;
    double premium;
public:
    policy();
    ~policy();
    void renew();
    bool claim(double);
}
```

The constructor function policy() is called automatically every time you define an instance of the class policy, such as myPolicy. The constructor does whatever is involved in setting up myPolicy as an open policy object *immediately after* myPolicy is created. When the myPolicy instance is destroyed or goes out of scope, usually at the end of a function, the destructor function ~policy() is automatically called. The destructor does whatever is necessary to de-initialise the instance myPolicy immediately before it is destroyed on exit from the function. Here's an example where the constructor and the destructor are called in turn:

```
void clientFunc()
{
    policy herPolicy;       // define class instance
                            // also quietly call constructor policy()
    //   do some processing

    //   destructor quietly called here to tidy up
}
```

You can specify constructor functions with arguments, but not destructors. We'll see more of this in Chapter 9, *Class services*.

Overloading

C++ provides two kinds of overloading: *function overloading* and *operator overloading*. This section gives an example of both, using the policy class.

Using function overloading, you can use more than one version of a function with the same name. The appropriate version is called according to the parameter types used by the function. Using operator overloading, you can make a standard C++ operator, such as + or -, take on a new meaning.

Here is the policy class declaration changed to include an overloaded function and an overloaded operator:

```
class policy
{
private:
    char name[30];
    char address[50];
    char polno[8];
    double ins_value;
    double premium;
public:
    policy();
    ~policy();
    void renew();
    void renew(double newPrem);
    bool claim(double);
    bool operator-=(double claimAmt);
};
```

The function renew is overloaded. Prototypes of two versions of it are declared. The appropriate version is selected depending on the absence or presence of arguments in the function call. If you wanted to specify a non-default premium amount in renewing the policy, you could call the second renew member function like this:

```
myPolicy.renew(500.00);
```

The standard C++ operator -= (subtract whatever is on the right-hand side from the variable on the left) is overloaded to provide an alternative way of claiming money on the policy. The keyword operator announces that the -= operator is to be given a special meaning when it is used with an instance of the class policy. operator-= is itself a function declaration, specifying a return type of bool.

Here are possible definitions of the overloaded renew and operator-= functions:

```
void policy::renew(double newPrem)
{
    premium = newPrem;
}

bool policy::operator-=(double claimAmt)
```

```
    {
        // subtract claimAmt from claim fund

        return(true);
    }
```

If you call the renew function with one argument:

```
    int main()
    {
        policy hisPolicy


        hisPolicy.renew(250.00);

    }
```

then you get the instance of the function shown above. If, on the other hand, you'd like to do a special claim, you could do this

```
    hisPolicy-=20000   // now emigrate!!
```

When the compiler sees this special use of the -= operator in the context of an instance of the policy class, it "knows" to call the member function operator -= as shown above. So, although this superficially looks like a subtraction from the class instance hisPolicy, it's really just a call to a member function of that instance.

Inheritance

Class inheritance is one of the main characteristics of the OOP approach. If you have a base class, you can also declare a derived class that takes on all the attributes of the base and adds more. The derived class is said to inherit the base class. You can build derived-class hierarchies of arbitrary depth.

Single inheritance occurs when a derived class has only one base class; *multiple inheritance* is when a derived class has more than one base class. You'll see multiple inheritance in Chapter 10.

Here is a simple example of single inheritance based on the policy class:

```
    class policy
    {
    protected:
        char name[30];
        char address[50];
        char polno[8];
        double ins_value;
        double premium;
    public:
        policy();
    ~policy();
    void renew();
    void renew(double newPrem);
    bool claim(double);
    bool operator-=(double claimAmt)
}
```

The keyword private, which might be expected in the base class policy, is instead protected so that its characteristics can be inherited by the derived class motor. Member functions of derived classes are allowed access if protected is used. Now you can declare the derived policy class, motor:

```
class motor : public policy
{
private:
    double reserve;
public:
    void no_claims_bonus();
};
```

motor inherits all non-private data and function members of the base class policy. motor adds the data member reserve. A member function of motor can now directly access any of the data members of policy. None of these data members can be accessed directly by code other than member functions of the class hierarchy. Member functions of the base class can't access members of the derived class.

motor inherits all member functions of the base class. If, in motor, inherited functions are redeclared, those redeclarations are said to *override* the inherited functions. Inherited functions, however, need not be overridden; they may be declared for the first time in a derived class and join inherited data and functions as members of the derived class.

The main function defines p1 and p2 as objects of type policy and motor respectively:

```
int main()
{
    policy p1;
    motor p2;

    // Calls here to 'policy' and 'motor' member
    // functions

}
```

The C++ I/O system

The C++ environment provides the *Stream I/O* library. This is called Stream I/O and is based on the declarations contained in the header file iostream. This section introduces some of the simple facilities offered by Stream I/O.

The iostream header file overloads the shift operators >> and << to be input and output operators. You can use these operators with the four standard input and output streams, which are

cin	Standard input stream
cout	Standard output stream
cerr	Standard error stream
clog	Buffered equivalent of cerr, suitable for large amounts of output

The standard input stream typically represents the keyboard, the standard output stream the screen. cin is of type istream, a class declared in iostream. The other three streams are of type ostream, also declared in iostream.

If you define four variables

```
char   c;
int i;
float  f;
double   d;
```

you can display their values by inserting on the output stream:

```
cout << c << i << f << d << endl;
```

You can read from the input stream in much the same way

```
cin >> c >> i >> f >> d;
```

Here are two program examples showing some other C++ I/O facilities.

```
#include <iostream>
using namespace std;

int main()
{
    char c;
    while (cin.get(c))
        cout.put(c);
}
```

The get member function of class istream extracts one character from the input stream and stores it in c. The put member function of class ostream inserts one character on the output stream.

```
#include <iostream>
using namespace std;

const int MAX = 80;

int main()
{
    char buf[MAX];

    while (cin.getline(buf, MAX))
    {
        int chars_in;
        chars_in = cin.gcount();
        cout.write(buf, chars_in);
    }
}
```

getline extracts at most MAX - 1 characters from the input stream and stores them in buf. getline by default finishes extracting characters after a newline is entered.

gcount returns the number of characters extracted by the last call to getline.

cout.write inserts at most chars_in characters on the output stream.

The effect of these two programs is to copy characters and strings from standard input to standard output (keyboard to screen).

Your first real C++ program

You now know enough about C++ to understand a non-trivial program and to get it working.

Here's program implemented with classes in C++ that provides a simple model of the operation of a bank account. It is organised in three files: the header file accounts.h, the function program file accfunc.cpp, and the main program file accounts.cpp which acts as a 'driver' for the functions declared as part of the class cust_acc. First, the accounts.h header file:

```
class cust_acc
{
private:
     float bal;
     int acc_num;
public:
     void setup();
     void lodge(float);
     void withdraw(float);
     void balance();
};
```

accounts.h declares the cust_acc class, which has two private data members and four public member functions. The definitions of those member functions are given in accfunc.cpp:

```
/*
 *   Program file 'accfunc.cpp'
 *   defines 'cust_acc' member functions.
 */
#include <iostream>
using namespace std;

#include "accounts.h"

//
//   customer_account member functions
//
void cust_acc::setup()
{
     cout << "Enter number of account to be opened: ";
     cin >> acc_num;
     cout << "Enter initial balance: ";
     cin >> bal;
     cout << "Customer account " << acc_num
               << " created with balance " << bal << endl;
}
void cust_acc::lodge(float lodgement)
{
    bal += lodgement;
    cout << "Lodgement of " << lodgement << " accepted" << endl;
}

void cust_acc::withdraw(float with)
{
    if (bal > with)
    {
        bal -= with;
        cout << "Withdrawal of " << with
             << " granted" << endl;
        return;
    }
    cout << "Insufficient balance for withdrawal of " << with << endl;
    cout << "Withdrawal of " << bal << " granted" << endl;
    bal      = (float)0;
}

void cust_acc::balance()
{
    cout << "Balance of account is " << bal << endl;
}
```

Finally, here is the accounts.cpp program file. It contains a main function that acts as a 'driver' of the functions declared in the class cust_acc. In the simulation, an account object called a1 is created. The function setup is called immediately after the creation to initialise the object in memory. This is done by prompting the program's user for initial balance and account-number values. Then, an amount of 250 is lodged to the account and 500 withdrawn. The account balance is reported after each of these operations.

```
/*
 *  Program file 'accounts.cpp'
 *  drives the 'cust_acc' class
 */

#include <iostream>
using namespace std;

#include "accounts.h"

int main()
{
    cust_acc a1;
```

```
        a1.setup();
        a1.lodge(250.00);
        a1.balance();
        a1.withdraw(500.00);
        a1.balance();
}
```

The header file iostream is included in both accfunc.cpp and accounts.cpp. It contains, among other things, all declarations necessary to allow use of the input and output streams cin and cout. accounts.h is also included in both files, making the class declaration of cust_acc visible throughout the program.

The four member functions of the class cust_acc are called from main, in each case being qualified by the class object a1. To call the functions from main without qualification would result in compilation errors. The data members of cust_acc can only be used within those functions.

The bank account program is a very straightforward use of classes and the object-oriented programming approach. Using the Borland C++ Builder 5 compiler, I built it using the command-line

```
bcc32 accounts.cpp accfunc.cpp
```

producing the executable program accounts.exe. To execute the program, use the command-line

```
accounts
```

You should yourself enter and build the program. When you run it as shown, you should get results like these (bold indicates values entered by the user):

```
Enter number of account to be opened: 12345
Enter initial balance: 750
Customer account 12345 created with balance 750
Lodgement of 250 accepted
Balance of account is 1000
Withdrawal of 500 granted
Balance of account is 500
```

Summary

We have travelled at warp speed through many of the essential constructs of the C++ language. So far, I've simply not dealt with a number of important aspects of it. The focus is on rushing you along the short path to minimal competency in C++ programming. Then you will be more ready to face the other 90% of the language. Concepts you will have to understand include these, which are covered in the remaining chapters:

- The 'C language subset' in Chapters 2 to 7
- Initialisation and assignment
- Advanced overloading
- Friends
- Virtual functions
- Single and multiple inheritance
- Templates

Although there's a lot you'll have to know about these topics to be a fully-fledged C++ developer, most of them are, so to speak, variations on themes already stated in this chapter. The most important thing for you do to do is to 'get a handle' on the way I've presented C++ in this chapter.

Exercises

1. Design and implement a C++ class to hold data and operations pertaining to a car object. There should be at least four data members: weight, length, colour and the maximum speed. Member functions could include start, stop and accelerate operations, as well as turn and reverse.
2. Extend the class cust_acc with a derived class called savings. This should include interest and calc_interest data and function members while inheriting everything in cust_acc. Implement the calc_interest function in accfunc.cpp.
3. Include an overloaded += operator in cust_acc to provide an alternative to the lodge function for adding money to the account's balance.

2 How C++ handles data

Basic data types and qualifiers

C++'s data types

To be able to use the power of C++ effectively in your programs, you need to know more about the ways in which the language represents data. As you can see in Chapter 1, there are five simple data types in C++, which are used as type specifiers in the definition of variables:

bool	true/false value; size dependent on system
char	usually a single byte, storing one character
int	an integer of a size dependent on the host computer
float	a single-precision floating-point (real) number
double	a double-precision floating-point (real) number

You can qualify the simple data types with these keywords:

signed	long	unsigned	const
short	volatile	mutable	

On computers for which the 8-bit byte is the smallest addressable memory space and therefore the basic data object, the char type specifies a variable of one byte in length. char specifies enough memory to store any member of the local system's *character set*.

On a computer with a 32-bit processor (32-bit addressing and integer size) and any of the Windows operating systems since late versions of Windows 95, the default integer size is 32 bits. The same is true of most UNIX variants, although there are now some 64-bit UNIX implementations. In this book, I assume 32-bit processor and integer size. Given this assumption, an int in a C++ program is 32 bits (4 bytes). A float is also usually implemented in 32 bits, while a double takes up 64 bits or eight bytes.

You can combine the basic types with the qualifiers listed above to yield types of sizes varying from the defaults. The table below gives possibilities for combination of the basic data types and the qualifiers.

Qualifier\Type	char	int	float	double
signed	X	X		
unsigned	X	X		
short		X		
long		X		X
const	X	X	X	X
volatile	X	X	X	X

The default integer type is signed int. If the leftmost bit in the (32-bit) integer bit-pattern is 1, the number is treated as negative; 0 indicates positive. The types int and signed int are synonymous. unsigned int forces the integer value to be positive. The sign-bit is not used and it is possible to accommodate in an unsigned int a positive value twice as large as for an ordinary int.

With an int size of 32 bits, short int is generally 16 bits and long int is 32 bits. You can simplify short int to just short, unsigned int to unsigned and long int to long.

In addition to the possibilities listed in the table above, signed short int and unsigned short int are both legal, as are signed long int and unsigned long int.

const, volatile and mutable qualifiers

The qualifier const may be prefixed to any declaration, and specifies that the value to which the data object is initialised cannot subsequently be changed. In normal C++ programs, the const qualifier is widely used; volatile and mutable are much rarer.

The qualifier volatile informs the compiler that the data object it qualifies may change in ways not explicitly specified by the program of which it is part.

For ordinary variable definitions, many C++ compilers are allowed to carry out optimisation on the assumption that the content of a variable does not change if it does not occur on the left-hand side of an assignment statement. The volatile qualifier causes the suppression of any such optimisation.

For example, volatile could qualify the definition of a variable, as in:

```
volatile int clock;
```

The value of clock might be changed by the local operating system without any assignment to clock in the program. If it were not qualified volatile, the value of clock might be corrupted by compiler optimisation.

The mutable qualifier is used to specify that an attribute of a class or structure remains changeable, even if the instance of the class or structure is declared as const. There is more on this in Chapter 4.

Numeric capacities of data types

Here are some example declarations that assume a system with a natural 32-bit integer:

```
short x;          // x is 16 bits long and can hold integer values
                  // in the range -32767 and 32767

int y;            // y is 32 bits long and holds integer values in the range
                  // -2147483647 to 2147483647

long z;           // same as 'int' above
```

```
unsigned short a;   // sign bit disabled, can hold positive integer
                    // values up to 65535

unsigned b;      // 'int' definition with sign-bit disabled, can hold
                 // positive integer values up to 4294967295

float c;         // c is 32 bits long and can hold a fractional number in a
                 // floating-point form in the range 3.403 X 10^38 to
                 // 1.175 X 10^-38
double d;        // d is 64 bits long and can hold a fractional number in
                 // the range 1.798 X 10^308 to 2.225 X 10^-308
```

Here is a program, maxint1.cpp, which finds the largest possible numeric value that can be stored in an int on your computer:

```
/************************************************************************
 *
 *    'maxint1.cpp'      Program to find the largest number that can
 *                 be stored in an 'int' on this computer
 *
 ************************************************************************/

#include <iostream>
using namespace std;
int main()
{
      int max, accum = 1;
      //    Increment accum in a loop until accum goes negative
      while(accum > 0)
    {
         max = accum;
         accum = accum + 1;
      }
      cout << "Maximum int value is " << max << endl;
}
```

This program does the job in the obvious but crude way: counting starts at 1 and the variable accum is repeatedly incremented by 1 until the sign bit changes and the integer goes negative. Just a few years ago, on 32-bit systems that could 'only' do ten million or so additions per second, this program took around 4 minutes of execution time to increment accum to its maximum signed value of more than two thousand million. But – an undoubted sign of progress – my multi-Gigahertz PC can now run this program in 5 seconds (more than 400 million while-loop iterations, assignments and additions per second!).

Fast as the new hardware is, maxint1.cpp remains crude and 'brute-force'. In addition, as soon as the normal integer size goes to 64 bits, the problem 'goes off the scale'. The maximum signed integer size that can be stored in 32 bits is

2,147,483,647 – divide by 400 million loops per second and you can see that maxint1.cpp executes in about 5 seconds. The maximum signed integer size for 64 bits is, however, 9,223,372,036,854,775,807! At 400 million loops per second, it will take more than 731 years to increment accum by steps of 1 to a negative value. So, we need a better way. The program maxint2.cpp provides it, in a good example of how C++ lends itself to clever 'bit-level' programming.

```
/**********************************************************************
 *
 *     'maxint2.cpp' —   Program to find the largest number that can
 *                  be stored in an 'int' on this computer
 *
 **********************************************************************/

#include <iostream>
using namespace std;

int main()
{
      int shift = 1, accum = 0;

      //     loop until a further shift would set the sign bit

      while(shift > 0)
      {
            //     add shift to the accumulator and double it
            accum = accum + shift;
            shift = shift * 2;
      }
      cout << "Maximum int value is " << accum << endl;
}
```

Instead of many billions of repeated additions, maxint2.cpp relies on repeated multiplication, each time by two. Each multiplication shifts the leftmost bit leftwards as the (binary) number gets larger in the sequence 1, 10 (decimal 2), 100 (decimal 4), 1000 (decimal 8) and so on. Even on a 64 bit system, after just 64 multiplications, the leftmost (64th and sign) bit of the number is set to 1. At that point, the number goes negative and the maximum value is left in accum. Regardless of the size of integer being dealt with, maxint2.cpp executes in a split-millionth of a second.

When you run either maxint1.cpp or maxint2.cpp, the output will be something like this:

```
Maximum int value is 2147483647
```

with the great difference in efficiency between the two versions noted. This output indicates, by the way, that I ran the program on a 32 bit system (DOS virtual machine under Windows XP, to be specific).

Initialisation and assignment

When you define a variable in a C++ program you should assume that it will initially be set to a garbage value You should therefore initialise variables, where necessary, when you define them

In maxint.cpp the variables shift and accum are initialised as part of their definition

```
int shift = 1  accum = 0;
```

Initialisation means that a variable is set to a value at the point of definition, *assignment* separates the setting of value from the definition

```
int shift;
shift = 1
```

You can initialise a variable of type long int like this

```
long big_num = 1000000L
```

The trailing L explicitly tells the compiler that the 1000000 is to be a long integer

A variable of type char can be initialised to a character (or numeric) value

```
char c = 'a';
char d = 97  // same thing: 97 is ASCII 'a'
```

Expression type

Every expression has a type If the expression contains an assignment its type will be that of the variable being assigned to: if not the expression will be of a type determined by its constituent parts Ideally in a given assignment expression, all the variables and data should be of the same type

```
int result;
int a = 5.
int b = 6

result = a + b + 7;  // value of result becomes 18
```

Here all the variables and data are of type int (the literal 7 is implicitly of type int), the additions and assignment are straightforward. Sometimes, however it isn't so easy

```
int result,
int a = 5
double b = 6.357291

result = a + b + 7;
```

The expression a + b + 7 is of type double   in effect the highest common denominator of the types of the three operands The result of the addition is 18 357291 but this is truncated across the assignment to 18 Because the type of result is int the type of the expression (double) on the right hand side of the assignment is forced downwards to match with corresponding loss of data

Type casting

If it were an ideal world, you would ensure that all variables you use in a given expression were of the same type. Then no conversions would be needed, for example between integer and fractional, or between character and integer quantities. But, as we know, life isn't that simple. Sometimes, to keep things correct, we must explicitly force conversions between data types. This operation in C++ is referred to as *type casting*.

Type casting is done using the *unary typecast operator*, which is a type specifier enclosed in parentheses and prefixed to an expression. The cast does not change the value of the expression but may be used to avoid the consequences of unintended type conversions.

Type conversion can be vital. Imagine you're calculating the total number of days that have elapsed since January 1, 1900. The computation would look like this:

```
days_total = (long)yy * 365 + no_leaps + days_year + dd
```

On a 16-bit integer system, the intermediate calculation yy * 365 exceeds the 32,767 integer size limit if the date is later than September 18, 1989. The intermediate calculation (long)yy * 365, forcing yy temporarily to be long, works on all systems, having a capacity of 2,147,483,647 in both 16- and 32-bit environments.

In C++, the expression 5/7 gives zero, as a result of integer division. If you didn't want this, you could use the typecast operation:

```
(float)5/(float)7
```

to get the fractional result, .71428.

There are two available forms of unary typecasting as shown by the program oldcast.cpp:

```
#include <iostream>
using namespace std;

int main()
{
    double pi = 3.1415927;

    cout << pi << endl;
    cout << (int)pi << endl;
    cout << int(pi) << endl; // alternative
}
```

The mechanism of unary typecasting shown here originates with the earliest (early 1970s) definition of the C language. Over the years, the *old-style cast* (as it is sometimes known) has sometimes been badly used, effectively to switch off the otherwise strong C/C++ typing rules. As a consequence, the ISO C++ Standard has introduced into the language a number of explicit type-conversion operators. These are introduced later in this chapter, in the section *Type conversion*. You should be aware that old-style casting, while it still works, is considered to be *deprecated* (ISO for obsolete').

Naming conventions

Variables are defined or declared by association of a type specifier and variable name. For simple data objects, declarations and definitions are usually the same: it's enough for now to say that all definitions are declarations but that the converse is not true.

There are in C++ some simple rules concerning the names that may be used for variables. These names are also called identifiers.

A variable name must not be one of the set of C++ keywords (*reserved words*). You should not use library function names as variable names.

A variable name is a sequence of letters and digits. Distinction is made between upper and lower case letters. The underscore character _ also counts as a character and should be used for clarity in variable names:

```
next_record_from_file
```

being more readable than nextrecordfromfile. Names with internal capitalisation have also become acceptable and good practice:

```
nextRecordFromFile
```

You shouldn't use punctuation, control and other special characters in variable names. Also, don't use the underscore character at the start of variable names. If you do, there may be a clash with the names of certain library functions. Variable names may be any length, but keep your variable names to a maximum of 31 characters; some C++ compilers may treat only the first 31 as significant, ignoring further characters.

Here are some examples of incorrect variable name definitions:

```
int bank-bal // Wrong! incorrect hyphen
int 1sttime   // Wrong! leading number
int new?acc // Wrong! invalid character
```

Arithmetic operations

In your C++ programs, you can perform calculations with these basic arithmetic operators:

+	addition	-	subtraction
*	multiplication	/	division
%	modulus		

I also introduce variants of these four operators that are very characteristic of C++ so that you understand them when you see them in program examples that follow:

++ add one to a variable, as in var++

-- subtract one from a variable, as in var--

+= add a quantity to a variable, as in var+= 5

-= subtract a quantity from a variable, as in var-= 7

Use of the division operator, /, with two or more integer operands causes integer division and consequent truncation.

3/5 equals zero

5/3 equals 1

The *modulus* or 'remainder' operator, %, may only be used with *operands* of type int or char. You can't use it with float or double operands. Multiplication, division and modulus operations are done before addition and subtraction. *Unary minus* operations (for example -(a + b), as opposed to the binary a - b) are carried out before any of these. You can see the *precedence* of the arithmetic operators from this series of assignments:

```
int x = 5;
int y = 6;
int z = 7;
int result;



result = x + y * z;       //    result == 47
result = y / x * z;       //    result == 7
result = (x + y) * z;     //    result == 77
result = -y * z + x;      //    result == -37
result = z / x % y;       //    result == 1
```

Finally, you may have noticed that there is no operator for exponentiation: you have no way of expressing something like x *to the power* 5. To do this, you must use the pow library function described in Chapter 14.

Here is an example program, called sum1ton.c, that implements the so-called Abelian series after the famous mathematician Abel. As a bright 10-year-old, young Abel and his class at school were set time-killing exercises by their teacher. One such was the job of adding all the numbers from 1 to 100. Abel, using his series was able instantly to present the result to his teacher. The series is described by the equation $t = n(n + 1)/2$, where t is the total and n is the number at the end of the series. Here's the program:

```
/*************************************************************************
 *
 *     sum1ton.cpp — Program to calculate the sum of all the integers
 *      in the range 1 to n, using the Abelian formula (n * (n + 1))/2
 *
 *************************************************************************/
#include <iostream>
using namespace std;

int main()
{
     int n, sum1toN

     cout << "Enter a number: ";
     cin >> n;

     sum1toN = (n * (n + 1))/2;

     cout << "Sum of the integer series 1 to "
          << n << " is: " << sum1toN << endl;
}
```

Try entering and building this program. When you run it, you are prompted to enter a number that will be the limit of the series to be summed. The input stream, cin, is used to read your answer from the keyboard. The separate characters of the number you entered ('1', '0', and '0' of the number 100) are collectively converted to the numeric form and are stored in n. Abel's formula calculates the sum of the series, which is displayed by the cout statement. Here's the expected display (user input in boldface):

```
Enter a number: 100
Sum of the integer series 1 to 100 is: 5050
```

Different kinds of constants

Every basic data object – bool, char, int, float, double – is a number. A number used explicitly, not as the value of a variable, is a *constant*. Constants are such things as the integer 14, the character 'a' and the newline '\n'.

The kinds of constants that you can use in C++ expressions include these:

Integer constants	Character constants
String constants	Floating-point constants
Special character constants	Enumeration constants

Integer constants

The *integer constant* 14 is a data object inherently of type int. An integer constant such as 500000 that on a 16-bit system is too large to be accommodated by an int is treated by the compiler as a long int.

An integer constant can be prefixed with a leading zero: 014 is interpreted as being of base 8 (*octal*) and equals decimal 12. An integer constant can have the prefix 0x or 0X:

```
0x14 or 0X14
0x2F or 0X2F
```

The compiler treats these constants as *hexadecimal* (base 16). Hexadecimal 0x14 equals decimal 20, 0x2F equals decimal 47.

Character constants

A character constant is a single character, written between single quotes: 'a'. A character constant is a number. After the definition and initialisation:

```
char ch = 'a';
```

ch contains the numeric value decimal 97. Decimal 97 is the numeric representation of 'a' in the ASCII character set, which is used in nearly all PCs and UNIX systems. If a different character set is used (for example EBCDIC, used largely on IBM and compatible mainframes), the underlying numeric value of 'a' is different.

Another example: '0' (*character zero*) is a character constant with ASCII value 48. '0' has nothing at all in common with *numeric zero*, so after the definitions:

```
int n = 0;
char  c = '0';
```

the integer n contains the value zero, the character c contains the value 48.

To summarise: a character is represented in C++ by a small (in the range 0 to 255) number that corresponds with the position reserved for that character in the character set being used. If that number (say 98) is interpreted as a character, it is treated as the letter 'b'. Interpreted as a number, the character may be used in arithmetic like any other number.

String constants

You specify character constants with single quotes. *literal string constants* by contrast use double quotes:

```
"This is a string constant"
```

A string constant is also known as a *string literal*. The double quotes are not part of the string literal, they only delimit it.

Floating-point constants

Floating-point constants are always shown as fractional and can be represented in either normal or *scientific notation*

```
1.0
335.7692
 .00009
31.415927e-1
```

Floating point constants are of type double unless explicitly suffixed with f or F, as in

```
1.7320508F
```

which is of type float.

Special character constants

The newline character '\n' is a character constant. There is a range of these special character constants – also known as *escape sequences*.

They are:

```
\n  // newline
\r  // carriage-return
\t  // tab
\f  // formfeed
\b  // backspace
\v  // vertical tab
\a  // audible alarm - BEL
\\  // 'escape' backslash
\?  // 'escape' question-mark
\'   // 'escape' single quote
\"   // 'escape' double quote.
```

The escape sequences are used in place of the less intuitive code-table numeric values. In cout statements, '\n' is sometimes used at the end of the format string to denote advance to a new line on the standard output device (note that the manipulator endl is preferred). You could instead use the equivalent ASCII octal numeric code '\012' but this is less intuitively clear, as well as not being portable to systems using code tables other than ASCII.

Other characters can also be *escaped out*. Use of the lone backslash causes any special meaning of the following character to be suppressed. The following character is treated as its literal self. For example, the statement

```
cout << "This is a double quote symbol: \"" << endl;
```

causes this display on the standard output

```
This is a double quote symbol: "
```

There are many other special characters which do not have an identifying letter and are represented by their number in the character set, delimited by single quotes. These are examples from the ASCII character set

```
#define   SYN   '\026'     // synchronise
#define   ESC   '\033'     // escape
```

This is a good use of the preprocessor, equating symbolic constants with numeric control characters. The symbolic constant ESC in the middle of a communications program makes more sense than '\033'

Here is a program, charform.c, that shows how the contents of a char variable can be interpreted differently using the format codes of the C Library printf function

```
/*****************************************************************
*
*    'charform.cpp'     Program to show interpretation of a
*   character's value according to various
*   'printf' format codes
*
*****************************************************************/
#include <iostream>
using namespace std;

#include <cstdio>

int main()
{
     int c;

     printf("Enter a character: ");
     c = getchar();

     printf("Character %c, Number %d, Hex, %x, Octal %o\n", c, c, c, c);
}
```

The printf function uses the %-prefixed format codes to tell it how to interpret the variables following. %c means 'character', %d means '(decimal) number', and so on. The most important purpose of this program is to confirm that a character is no more than a number and that it can be interpreted as different kinds of numbers. Try running it yourself to confirm this. Here's what the display should look like. The character that I input for interpretation was the question mark.

```
Enter a character: ?
Character ?, Number 63, Hex: 3f, Octal 77
```

Enumeration constants

The enumeration constant is a list of integer constant values, for example

```
enum seasons {SPRING,SUMMER,AUTUMN,WINTER};
```

The four names in this example have values associated with them of 0, 1, 2 and 3 respectively, unless the programmer chooses to depart from the default:

```
enum seasons {
                    SPRING=1,
                    SUMMER=2,
                    AUTUMN=3,
                    WINTER=4
                 };
```

Having made either of the above declarations, you can define a variable associated with the enumeration constant and with type enum seasons:

```
enum seasons time_of_year;
```

time_of_year can only have the values SPRING, SUMMER, AUTUMN or WINTER. You can now do a test like this:

```
if (time_of_year == SUMMER)
   go_sunbathing();
```

Arithmetic operations on enumeration constants may be allowed by individual compilers but are illegal in the ISO C++ language definition.

Enumeration constants have limited application, but they are useful in a few specialised types of application program. In any situation where input data can be only one of several mutually exclusive types, the type can be recorded using an enumerated constant:

```
// enum might be used in a spreadsheet
enum inputType {INTEGER, FRACTION, STRING};

enum inputType dataType;

// set default data type
dataType = INTEGER;
```

Pointers and references

Pointers and addresses

A pointer is a data type; a variable of that type is used to store the address of a data object in memory. A variable definition allocates space for the data and associates a name with that data. The data name refers directly to the data stored at the memory location. Pointers, on the other hand, are data objects that point to other data objects.

You can define a character variable and a character pointer like this:

```
char c;
char *cptr;
```

cptr is a *pointer to* a data object of type char.

The statement

```
cptr = &c;
```

uses the *address-of* operator to assign the address of c to the character pointer cptr. After the assignment, cptr points to c. *cptr *dereferences* the pointer and is the *contents of* or the *object at* the pointer cptr. *cptr equals c. Note that it's always an error to dereference a pointer which has not been initialised to the address of a data object to which memory has been allocated.

You can define and use an *integer pointer* (a pointer that should be used only with variables of type int).

```
int i = 6;
int *ip = &i;
```

Taking the example of the integer pointer, you can reflect on the following truisms:

```
i == 6
ip == the address of the integer i
*ip == the object at the address: 6
```

A major confusion arises from the dual use of the *ip sequence. The definition of the pointer

```
int *ip = &i;
```

specifies that the variable ip is of type int *, or integer pointer. On the other hand, when the pointer is later used and dereferenced:

```
*ip
```

the object at the pointer ip (6) is retrieved. If you keep in mind the difference between the sequence *ip used when the pointer is being defined and *ip used to retrieve the value stored at the pointer, you will move a long way toward being fluent in the use of C++ pointers.

Pointers to data objects

You can use pointers with all data objects that may be defined in C++, including arrays, structures and other pointers. In this section, we're particularly interested in arrays, especially character arrays.

In the definition of the integer pointer above, the address-of operator & is used in the initialisation of the pointer ip with the address of the integer i:

```
        int i = 6;
        int *ip = &i;
```

The address-of operator must be used when initialising pointers, except when the address of an array is being assigned to a pointer of the same type as the array. In the sequence:

```
    char instring[50];
    char *cptr = &instring;
```

cptr is in fact initialised to the *address of the address* of the pointer. *An array name is the array's address*; to initialise the pointer with the array's address

```
    char *cptr = instring;
```

is all that is needed to do the job correctly. The fact that an array's name is its address while the name of any other variable is not is one of the quirks of the C++ language that causes most inconvenience even for experienced programmers.

Next, we define and initialise a character array and set a pointer pointing to the start of the array:

```
    char textline[50] = "Many a time and oft on the Rialto";
    char *cp = textline;
```

The value of textline[0] is the letter 'M', the value of textline[1] is 'a', and so on. Similarly, the value of *cp after the pointer has been initialised is 'M'. The value of *(cp+1) is 'a' and the value of *(cp+2) is 'n'. You will see more in Chapters 4 and 7 of pointers used with arrays.

Here is an example program, litptr.cpp, that uses pointers to traverse a character array one character at a time, displaying each character on the way.

```
/**********************************************************************
 *
 *    'litptr.cpp' —  Program to display each character in a literal
 *                    string using a simple character pointer
 *
 **********************************************************************/
#include <iostream>
using namespace std;

int main()
```

```
{
        char litstr[50] = "The quality of mercy is not strained";
        char *start, *p;

        start = p = litstr;

        while(*p != '\0')
        {
                cout << *p;
                p++;
        }
        cout << "\nString is " << p-start << " chars long" << endl;
}
```

Apart from wondering who said "The quality of mercy ..." and in what Shakespeare play, you can examine the pointer technique shown by the program. The array litstr is initialised at the point of its definition with the literal string "The quality of mercy ...". The contents of the array are now a C string terminated with a null character '\0', implied by the double quotes in "The quality of mercy ...". Both the pointers start and p are set pointing to the start of litstr. Then, while the contents of p are not the null character, all the characters in the array are displayed individually, along with the length of the string:

```
The quality of mercy is not strained
String is 36 chars long
```

In case you think that this stuff with pointers might only be used by nerds and code-freaks, be disabused of the notion now. Text processing, and other use of pointers, is central to C++ programming. To be a good C++ programmer, you have to be good at pointers. On that consoling note, be happy that you have now surmounted one of the steeper obstacles presented by the C++ language.

References

C++ provides an alternative to the system of pointer initialisation and dereferencing shown above. This is the *reference* type. A reference is not a copy of the variable to which it refers, neither is it a pointer (although, 'under the covers', the C++ system may implement references with pointers). You should think of a reference as an *alias* for the name of a variable, one which, when set (or 'seated') cannot be re-set. References are used mostly when passing arguments to functions – more in Chapter 3 – but, for now, here's a simple use of the reference type:

```
#include <iostream>
using namespace std;
int main()
{
        int n1 = 7;
        int n2 = 8;
        int &ref_n    = n1;

        ref_n *= 5;
        cout << ref_n << " " << n1 << "\n";
    }
```

In this program, the reference ref_n is assigned (made an alias for) the integer variable n1. The numeric quantity referenced (7) is multiplied by 5 and the result displayed. The ampersand (&) is used to denote ref_n as a reference. This usage is unfortunate because, as you've seen, the & also acts as the address-of operator during pointer assignment. This double-use of the ampersand symbol is confusing, and you just have to learn the difference according to the context in which the & is used.

When a reference object is used, no pointer-like de-referencing is needed: you just use ref_n, not *ref_n. This simplification of syntax is a point in favour of references but, for many, the dual-use ampersand confusion is an unfortunate price to pay.

The C++ 'string' class

Generations of C and C++ programmers – your current author included – acquired great expertise in manipulation of C strings of type char * along the lines shown in the (sample) string manipulation program cptr.cpp in the last section. Countless wet afternoons were spent on the delights of direct handling of strings using char pointers: finding, searching and replacing. The Standard C Library provides some useful functions for string handling; examples include strlen to find the length of a C-string and strstr to find the position of one string of text within another.

With the advent of C++, the string handling experts used the class construct to encapsulate the C-string. Now, strings could be assigned using the overloaded = operator and joined together (concatenated) using the overloaded +=. Member functions of the encapsulating class could search, replace and find substrings within strings. There was no limit to the extent of the possible refinements; the overriding objective was to present a high-level interface to the programmer using the class, hiding the nasty string pointer details. In this way, reliability, reusability, saved time and all the other benefits of object-orientation would accrue. Many class libraries were developed and marketed that also included classes for handling strings.

ISO C++ took even this diluted form of fun away. The Standard C++ Library now supplied an extremely comprehensive string class (in fact a *template*, but more of that later). There is no longer any need for the C++ programmer directly to handle character strings with pointers: the standard string class provides all the functionality you'll need and probably a lot more. Have no fear though: C strings and the possibility of writing string classes from scratch are still available in case you're pining for them on one of those wet afternoons. It's just that you don't have to use them any more.

I could write an entire book on the string class. The string class is, in fact, just an aspect of a C++ template called basic_string, which is to be found in the STL and which is in turn part of the C++ Standard Library. The STL doesn't provide support just for strings, but for *containers* in general. A string is a container of information in the same way as lists (like a list of names in a phone book), sets (any unordered collection) and queues (first in, first out, like a bus queue) – even though the different kinds of container behave slightly differently in detail. For example, inserting a character into a string is the same fundamental type of operation as is adding a member to a queue but, under the covers, the operations differ. The STL hides all these details and differences from you and allows you to use the different kinds of containers in the same way. From all this you probably (and correctly) get the sense that a *Made Simple* book of this size cannot exhaustively cover the string class, let alone the STL. This section provides a quick introduction, with more information given in Chapter 12 (*The Standard Library*).

Here is a program called string1.cpp, which sets up a few instances of the string class and does some basic operations.

```
#include <iostream>
using namespace std;
#include <string>
int main()
{
    string s1 = "Now is the time\n";
    string s2;
    string s3("to come to the aid\n");
    s2 = "for all good men\n";
    cout << s1 + s2 + s3 << endl;
    s1 += s2 += s3 += "of the party";
    cout << s1 << endl;
}
```

Notice first that the string header file is #included for the sake of its declaration of the standard string class and attributes. The main function defines three instances of the string class and in various slightly different ways (all legal and of similar effect) initialises them with strings of text. The line

```
cout << s1 + s2 + s3 << endl;
```

uses the overloaded + operator (defined within the standard string class. You don't have to worry about how) to concatenate the three lines of text. In a further operation:

```
s1 += s2 += s3 += "of the party";
```

concatenates and assigns to s1 all of the text, plus a new line added on. The displayed result of the program's execution is this:

```
Now is the time
for all good men
to come to the aid

Now is the time
for all good men
to come to the aid
of the party
```

The advantage of using the standard string class is that you just don't have to be concerned with how all the underlying assignments and concatenations are done. You just do the operations at an intuitive level using +, = and += operators and leave the 'system' to do the rest. The next example is a development of the above program, called string2.cpp.

The string instance is set to the text of the same verse as is the case in string1.cpp. A number of operations provided as standard by the string class are then performed: we find the length of the text in terms of the number of characters, the position in

```
#include <iostream>
using namespace std;
#include <string>
int main()
{
    string s1 = "Now is the time\n";
    string s2;
    string s3("to come to the aid\n");

    s2 = "for all good men\n";
    s1 += s2 += s3 += "of the party";

    cout << s1 << endl << endl;
    cout << "Total string length " << s1.length() << endl;
    cout << "Position of string " << s1.find("men") << endl;

    s1.replace(s1.find("men"), 3, "people");

    cout << endl;
    cout << s1 << endl;
}
```

the text of a particular substring, and, finally, replace that text with something more politically correct. Here is the output that is displayed on execution of string2.cpp:

```
Now is the time
for all good men
to come to the aid
of the party

Total string length 64
Position of string 29

Now is the time
for all good people
to come to the aid
of the party
```

The total interface provided by the standard string class — the full set of possible operations — is very large and is beyond the scope of this book to describe. For more information, you could do worse than look at my own *The C++ User's Handbook* (Butterworth-Heinemann, 2003) or *The C++ Standard Library*, Josuttis (Addison-Wesley, 1999).

Type conversion

The traditional C/C++ typecast mechanism is explained earlier in this chapter. This section introduces the new, stricter, typecasting operators made available with Standard C++. They are:

const_cast <type> (expr)	remove const variable qualification
dynamic_cast <type> (expr)	runtime determination of type of object pointed to; used mainly with RTTI (see Chapter 12)
reinterpret_cast <type> (expr)	interpretation of bit patterns
static_cast <type> (expr)	explicit typecast; replaces unary typecast

I'll look at two of these here. The dynamic_cast operator is used with polymorphism and run-time type identification and is covered in Chapter 12. reinterpret_cast is really beyond the scope of a *Made Simple* book. In summary, it forces the compiler to attempt conversion of types that are very 'far apart', such as pointer-to-double (double *) and pointer-to-char (char *). Doing this involves manipulation by the compiler of the actual bit-patterns of the pointer values. Results tend to be unpredictable and are *implementation-defined* (not portable from one C++ environment to another).

The conversion operator static_cast is used for explicit conversion operations that can be done by the compiler implicitly, although perhaps generating warning messages. Here's a program, static.cpp, that illustrates its use:

```
#include <iostream>
using namespace std;

int main()
{
    int result;
    int a = 5;
    double b = 6.357291;

    cout << "Intermediate result " << a+b+7 << endl;

    cout << "Typecast intermediate result "
        << a+static_cast<int>(b)+7 << endl;

    result = a + static_cast<int>(b) + 7;
    cout << "Truncated result " << result << endl;
}
```

The program's displayed output is this:

```
Intermediate result 18.3573
Typecast intermediate result 18
Truncated result 18
```

Constant (const) declarations can be overridden temporarily by using the const_cast conversion operator. If you define two pointers:

```
        char *cp;
        const char * ccp;
```

And set them both pointing to something, it is illegal to try to convert the const qualified pointer to a non-const:

```
        cp = ccp;
```

The example program const.cpp shows how to make the conversion:

```
    #include <iostream>
    using namespace std;

    int main()
    {
        char sig[50] = "Now is the time";
        const char *ccp = sig;

        cout << "Constant C-string is " << ccp << endl;

        char *cp;

        cp = const_cast<char *>(ccp);
        cout << "Non-const C-string is " << cp << endl;
    }
```

The conversion of ccp to cp is legal as shown.

You should be careful with mixing types and converting them, whether with the old-style casts or with the new conversion operators. Types, and the unwillingness of C++ to convert them with abandon, are there for a reason: to reduce program bugs caused by data and type conversions. It's best, if possible, not to convert at all; make all the variables of an expression the same type. If you must convert types, you should be aware that you're doing so, and of the possible consequences — double-to-int truncation like that shown in static.cpp above can cause its own problems. If you plan to convert, don't hope for the best and let the compiler do it implicitly. Make the conversion explicit, either with old-style casts or (preferably) with the cast operators.

Exercises

1 Write a program that defines and initialises an instance of the standard string class. The initialising text should contain a pattern that repeats at least once. The code of your program should search for and replace all occurrences of this pattern.

2 Write a program that displays the fully-qualified file pathname C:\MSVC\BIN. If you're using UNIX and prefer forward slashes, just humour me and do it with backslashes anyway.

3 Write a program that displays two literal strings, delimited by double-quotes in this way: "String1" "String2"

4 Write a program that defines a double variable to store the number 1.732050808 and then uses a loop to multiply the number by itself three times. Display the result. What do you get?

5 Write a program that defines two char variables, initialises them with the letters 'h' and 'g' respectively. Add the variables together and display the result as a character. What do you get?

3 C++ building blocks

Organisation of a C++ program

Every C++ program is organised as a number of functions (including exactly once main). Functions can belong to (be in the *scope* of) classes, namespaces or can be global. All the functions that make up a C++ program are stored in at least one *program file or translation unit*. Each program file must (on PCs at least) have a file name suffixed with .cpp. There can be as many program files as you like collectively making up a whole C++ program. Here is a program made up of two program files. The program doesn't do very much – each function except main makes a one-line display – but it is contrived to show the typical layout of a modern C++ program made up of multiple program files. If you understand this, you'll be able to read and write much more complex programs that, in spite of their complexity, obey the same layout rules. The first program file is **skelf1.cpp**

```
/*      Program file (translation unit) skelf1.cpp               */
#include <iostream>
#include "skelhead.h"
void c::func1()
{
    std::cout << "In class func1()" << std::endl;
      func1();
}
int main()
{
    c c_inst;
    c_inst.func1();
    func2();
    func3();
}
void func1()
{
    std::cout << "In global func1()" << std::endl;
}
```

The second translation unit is **skelf2.cpp**

```
/*      Program file (translation unit) skelf2.cpp               */
#include <iostream>
#include "skelhead.h"
void func2()
{
    std::cout << "In global func2()" << std::endl;
}
void func3()
{
    std::cout << "In global func3()" << std::endl;
}
```

You can see that the functions do not have to be in any special order. There is exactly one main function, as there must be, regardless of the number of program files. Any function may call any other, subject to the restrictions of global, namespace and class scope. Functions must not be *nested*. You can't define a function within a function. Function prototypes and other necessary declarations are are stored in the header file skelhead.h

```
class c
{
private:
public:
     void func1();   // prototype of c::func1()
};
void func1();   // prototype of global func1()
void func2();   // prototype of global func2()
void func3();   // prototype of global func3()
```

Let's start with the header file. It contains the declaration of the class c, which in turn has a prototype for the member function func1. The header file also contains declarations for the three global functions func1, func2 and func3. skelhead.h is #included in both of the program files, skelf1.cpp and skelf2.cpp, making all its declarations available to both.

The file skelf2.cpp contains the definitions of the global functions func1 and func2. skelf1.cpp contains the code for all the other functions, including main and the one func1, in the scope of the class c:

```
void c::func1()
{
    std::cout << "In class func1()" << std::endl;
      func1();
}
```

The first line specifies that this is the version of func1 that belongs to the class c. As noted in Chapter 1, the operator :: is called the scope resolution operator. It specifies the scope to which the function belongs, in this case the scope of the class c. By contrast, the function definition:

```
void func1()
{
    std::cout << "In global func1()" << std::endl
}
```

is in global or default scope and therefore does not have to have its scope resolved with the :: operator. The call to c::func1 from the main function is this:

```
c_inst.func1();
```

Prefixing the function name with an instance of the class c specifies explicitly that the version of func1 that is to be called is the one belonging to the class, not the global func1. To call the latter from main, you would use the simpler syntax

```
func1()
```

In fact, within c::func1, we want to make a call to the global version of func1. That's what the following line does:

```
  ::func1();
```

This use of the scope resolution operator tells the compiler to go 'outside' the scope of the class c to the enclosing (global) scope and call the global function func1.

A related point: all the programs up to now have started with the two lines

```
#include <iostream>
using namespace std;
```

In skelf1.cpp and skelf2.cpp, the namespace line is missing. The effect of this is that we must now explicitly specify the scope of the cout object and endl manipulator every time they are used:

```
std::cout << "In class func1()" << std::endl;
```

The prefix std:: is now needed for every use of any object in the Standard Library's scope. Considering the frequency with which such objects are used, the once-for-all specification

```
using namespace std;
```

is a convenience. In the above program, with no namespace specified, the one in force is the default (global) namespace that is supplied by the C++ run-time environment. In that global namespace, all functions are visible, but the library objects are not, hence the need for qualified specifications such as std::cout.

A final word on the syntax of header file inclusion:

```
#include <iostream>
#include "skelhead.h"
```

Any line in a C++ program that has as its first non-whitespace character the # (*hash*, *pound* to North American readers) symbol is processed, before compilation, by the C++ preprocessor. The contents of both header files are as a result included directly in the source code of both skelf1.cpp and skelf2.cpp, both of which files are then compiled. Use of angle brackets in the case of iostream tells the preprocessor to search for the file in the system standard directory (usually called INCLUDE). In the case of skelhead.h, use of double quotes adds the user's home directory and the current directory to the list of directories searched. With skelhead.h in the current directory and not in the system directory, the specification #include <skelhead.h> wouldn't work.

Lastly, why the .h suffix on skelhead and not on iostream? This is because the ISO standardising committee couldn't agree on what suffix to use and so agreed on none. Use of .h is still allowable.

Functions

A function is a sequence of C++ code executed from another part of the program by a function call. Every function's definition consists of two parts, the header and a compound statement. You should also specify a function prototype, or declaration. For the purpose of this section, I'm assuming, for simplicity, functions in global scope. In the previous section, there are two versions of the function func1

```
void c::func1(){}
void func1(){}
```

one of which is in the scope of the class c, the other being global. All the rules of function declaration, definition and specification apply to both equally, so I use the global form below.

Prototype

A prototype is an announcement to the compiler of the existence in the program of a function definition with a header matching the prototype. A function *prototype* means the same thing as a function *declaration* or a function *signature*. When the function is called after the compiler has seen the prototype, the compiler can check that the form of the function call is correct.

Although it is not always strictly necessary, you should always declare all your functions in advance to the compiler using a prototype:

```
int power(int, int);  // function prototype
```

You should make sure that the type of the return value and the number and types of arguments specified in the prototype exactly match those in the function header and in the call to the function.

Where you write many of your own functions, it's a good idea also to create your own header file (see skelhead.h in the previous section) and to store all the function prototypes there. Then you can #include the header file in all your program files, saving the bother of explicitly including the prototypes in every program file.

Header

A function header consists of a *return type*, the function name and the *parameter list*, also called an *argument list*, enclosed in parentheses. The terms 'argument' and 'parameter' are often used interchangeably, being strictly correct, the function call contains arguments and the function header parameters. In a function call, arguments are copied to matching parameters in the header of the function being called. A synonym for 'argument' in this context is *actual parameter*, while the term *formal parameter* can be used instead of 'parameter'.

Look at these definitions and function call:

```
int result, num, n;
result = power(num, n);   // function call
```

The header of the called function is this

```
int power(int num  int n)          // function header
```

In the header the first int is the function s return type This means that a value returned by the function across the assignment to result is an integer int is the default return type for functions power the function name is an identifier, as the term is defined in the section on naming conventions in Chapter 2 (page 42) In the function call the arguments num and n are copied to the parameters declared in the header of the called function power

The function starts at the first character to the right of the opening parenthesis in the header You can use the parameters enclosed in the parentheses inside the function in the same way as ordinary variables defined inside the function's main compound statement.

When you call the function, the values of the arguments used in the call to the function are copied into the parameters defined in the header When control is returned from the called function to the statement following the function call the values of the parameters are not returned. This means that an ordinary function in C++ cannot change the original values of arguments with which it is called.

If there are no arguments in the function call, their absence is in C++ explicitly specified in both the prototype and the function header by means of parentheses surrounding an empty argument list.

```
int power();          // prototype
result = power();     // call
int power()           // header
```

This tells the compiler explicitly that the power function takes no parameters If the call is made using arguments the compiler reports an error

Definition

A function s definition is its header followed by the body of the function (a compound statement) Here's a skeleton definition for the power function declared and called above:

```
int power(int num, int n)  // function header
{                          // function body
    int result

    // calculate num to the power n, assign to result
    return(result);
}
```

The last act of the function is to return the calculated result to the assignment in the function call Here is the full example program power cpp, which contains a real definition of the power function

```
/**********************************************************************
 *
 *     'power cpp' —    Program to raise numbers to a specified power
 *                      using a 'power' function.
 *
 **********************************************************************/
#include <iostream>
using namespace std;
long power(int, int),
int main()
{
      int num, n;
      long result,
      cout << "Enter number and exponent: "
      cin >> num >> n;
      result = power(num, n);
      cout << num << " to the power "
         << n << " is " << result << endl;
}
long power(int mantissa, int exponent)
{
      long result = (long)mantissa,
      while (exponent > 1)
      {
            result = result * mantissa,
            exponent--,
      }
      return(result);
}
```

Here the arguments num and n are copied from the function call to the parameters mantissa and exponent in power. mantissa and exponent are used within the function's body as easily as is the local variable result. When mantissa has been raised to the power exponent the value of the exponentiation is returned as result to the point at which the function was called. Notice that although exponent is repeatedly decremented in power the value of its corresponding variable n in main after the function call is unchanged. When you run the program, the display is similar to this

```
Enter number and exponent: 2 5
2 to the power 5 is 32
```

with user input in boldface. Typed input is accepted using the standard input stream cin.

Return values and parameters

A function has two ways of returning information to the calling function: by return value and by parameters. As in the example of the power function, the return statement is used to return a value from a function to where the function was called.

If you use return (without a value being returned) in a function, the result is unconditional return of control from the called function to the calling function. No particular value are returned from the function in this case. It is more usual to use return in a function to return control to the calling function with a value which is some use there. Typical uses are these:

```
return FALSE;
return(ERR_NO);
return result;
```

The parentheses surrounding the returned expression values are optional.

The alternative to return for sending back information from a called function is use of parameters. All arguments passed between functions in C++ are copied. A call to a function passing arguments by value (copying them to the parameters declared in the header) is known as a *call by value*.

C++ supports both call by value and *call by reference*. The latter is the means by which the values of arguments supplied in a function call are changed on return from the function. You can find more information on call by reference in the next section.

Here is a simple example, addnos1.cpp, of passing arguments to a function by value, returning the result to the point of call using a return statement:

```
/*******************************************************************
 *
 *    'addnos1.cpp' —  Program that calls a function to add two
 *                     numbers which sends back the result using
 *                     'return'
 *
 *******************************************************************/
#include <iostream>
using namespace std;

float add_nos(int, float);      // prototype

int main()
{
      int x = 14;
      float y = 3.162, sum;

      cout << "Numbers in: " << x << " " << y << endl;
```

```
        sum = add_nos(x, y);
        cout << "Sum of " << x << " and " << y << " is " << sum << endl;
}

float add_nos(int a, float b)
{
        return(a+b);
}
```

Calling a function to add two numbers smacks of overkill, but it effectively demonstrates the argument-passing mechanism.

First, using return, the called function add_nos returns its data, converted to type float, to the floating-point variable sum in main. add_nos is supplied with its data from the arguments x and y specified in the function call in main. These values are copied to the parameters a and b in the header of add_nos. You don't have to use different names for the arguments in the function call and the parameters in the function header. The latter could be x and y: they are named differently here for clarity only.

In main, x is defined as an integer and y as a floating-point value. Their values are copied to a and b in add_nos which are (and should be) defined with identical type. It's vital that the types of the corresponding arguments and parameters are the same: if they are not, the compiler will attempt automatic type conversion for you, not always with the results you might have expected. The arguments and parameters correspond by position and it's also essential that the order of the arguments is the same as that of the parameters. When you run the program, you get this:

```
Numbers in: 14 3.162
Sum of 14 and 3.162 is 17.162
```

Function call by reference

You've seen function call by value. Now we're going to look at call by reference. The distinction between the two types of call is very important. Call by value means that the values of the arguments sent by the calling function are copied to the parameters that are received by the called function. Call by reference means that the called function is using *the same data* (as opposed to a copy of the original) as that sent by the calling function.

Therefore, call by value does not change the values of the arguments in the calling function, no matter what is done to the parameters in the called function. Call by reference *does* change the values of the arguments in the calling function.

Call by reference with pointers

The program addnos1.cpp uses a return value to pass back the result of the function add_nos to the point of its call in main. Here is the equivalent program, addnos2.cpp, changed so that the computed sum is returned to the calling function as the changed value of the second argument supplied to add_nos.

```
/***********************************************************************
 *
 *    'addnos2.cpp'      Program that calls a function to add two
 *                   numbers which sends back the result using
 *                   pointers as parameters
 *
 ***********************************************************************/
#include <iostream>
using namespace std;

void add_nos(int, float *);      // prototype

int main()
{
      int x = 14;
      float y = 3.162;
      float z = y;

      cout << "Numbers in: " << x << " " << y << endl;
      add_nos(x, &y);
      cout << "Sum of " << x << " and " << z << " is " << y << endl;
}
void add_nos(int a, float *b)
{
      *b = *b + a;
      return;
}
```

As you can see, the type of y and b is no longer a simple float, but a pointer to float. The second argument supplied to add_nos is no longer just y, but the *address of* (which means the same as *pointer to*) y. The value copied to b is not the value of y, but a memory address for y. Altering the object at (or contents of) the pointer b in add_nos

```
*b = *b + a;
```

does not change the pointer b, but the object to which b is pointing, namely the value of the original y. The value of y displayed by the second cout statement in main reflects the change made by add_nos. The program's displayed output is the same as that of addnos1.cpp.

Array arguments

To have a function change the value of a data object supplied to it as an argument, the argument must be a pointer (or reference; more below) to the data object. In the case of arrays, the name of an array is its address. That address can be used as a pointer to the array's contents. It follows that, if an array name is used as an argument in a function call, the called function can change the object at the pointer (the array's contents) such that the change is seen in the calling function. In short, whenever you pass an array as an argument to a function, the contents of the array may have changed when control is returned from the called function.

Let's look at an example program, arrayarg.cpp, where a function, get_data, is called with three array arguments. The purpose of the function is to accept data from the standard input and place that data in the arrays.

```
/*
 *   arrayarg.cpp
 */
#include <iostream>
using namespace std;

void get_data(char [], char [], char []); // prototype

int main()
{
    char dd[5],mm[5],yy[5];

    get_data(dd,mm,yy);

    cout << "Day " << dd << " Month "
        << mm << " Year " << yy << endl
}
```

```
void get_data(char day[], char month[], char year[])
{
    cout << "Enter day: ";
    cin >> day
    cout << "Enter month: ";
    cin >> month;
    cout << "Enter year: ";
    cin >> year;
}
```

The *addresses of* the three character arrays are copied to the variable names in the function header get_data. The three arrays may then be used within get_data in the ordinary way. If the values of the array elements are changed, as they are here by user input, the change is reflected in the values of the arrays in main. The user's input-output sequence with this program is (input in boldface):

```
Enter day: 27
Enter month: 02
Enter year: 1974
Day 27 Month 02 Year 1974
```

C++ reference type

For many years, use of pointers and dereferencing, as in addnos2.cpp above, was the only means C programmers had of calling functions with reference arguments so that changes to their values would be reflected on return from the call. Although the pointer and dereferencing usage is the same as anywhere else in the language, it was felt necessary to simplify it somewhat. This was done by introduction of the reference type, referred to already in Chapter 2 under *Pointers and references*. Here is a program, addnos3.cpp, which does the same as addnos2.cpp, but with references instead of pointers:

```
/*******************************************************************
 *
 *    'addnos3.cpp' — Program that calls a function to add two numbers which
 *                    sends back the result using C++ references as parameters
 *
 *******************************************************************/
#include <iostream>
using namespace std;

void add_nos(int, float&);      // prototype

int main()
{
        int x = 14;
        float y = 3.162;
        float z = y;

        cout << "Numbers in: " << x << " " << y << endl;
        add_nos(x, y);
        cout << "Sum of " << x << " and " << z << " is " << y << endl;
}

void add_nos(int a, float &b)
{
        b = b + a;
        return;
}
```

The add_nos function prototype

```
void add_nos(int, float&);       // prototype
```

pinpoints the change. The type of the second parameter is now changed to float& (reference to float) from float * (pointer to float). The same change is made in the add_nos function header. Otherwise, the arguments used in the function call and the parameters used within the add_nos function now dispense with address-of and dereferencing operators and are used in their simple form. Many people prefer this simplicity and use of reference types, as opposed to pointers, is now preferred in C++ for function call by reference.

Storage class and scope

Every variable has a *storage class*. The storage class determines the *scope* (visibility) and *extent* (longevity) of the variable. It decides within how much of the program it is visible and how long it remains in being.

There are two storage classes, *automatic* and *static*. The C++ runtime system allocates and deallocates automatic storage during program execution. The compiler allocates static storage at compile time.

If you define a variable within a function, it's a local variable, also called an *internal variable*. The local variable is of automatic storage class and only exists for the duration of execution of that function. It's also only visible (or *in scope*) for the code of that function.

On the other hand, a variable defined outside all functions (an *external* or *global variable*) is of static storage class and exists for the duration of execution of the program. An external variable is in scope for all the code in all functions in your program.

A variable defined within a function and qualified with the keyword static is an internal variable but has static storage class. It also exists for the duration of the program's execution.

There are four storage class specifiers which you can use to specify explicitly a variable's storage class:

```
auto
register
static
extern
```

Auto and register specifiers

For a variable to be of automatic storage class, it must be defined within a function. All the variables we have so far defined within functions are automatic, or auto data objects.

This means that memory space for these variables is allocated each time the function is entered and that the space is discarded upon exit from the function. Because of this, an automatic variable cannot be accessed from any other function. The value of an automatic variable is lost on exit from the function in which it is defined. The integer definition

```
int x;
```

means the same thing as

```
auto int x;
```

if it is within a function and not otherwise qualified. However, auto is the default storage class specifier and need not be explicitly declared. (It rarely is.)

You can define a variable with the storage class specifier register. This is the same as auto in every way except that the compiler, on seeing the register specifier, attempts to allocate space for the variable in a high-speed machine register, if such is available.

static storage class

You can define a variable with static storage class by placing it outside all functions or within a class or function prefixed with the keyword static. A static variable has its memory allocated at program compilation time, rather than in the transient manner of auto.

A static internal (defined within a function) variable retains its value even on exit from that function. A *static internal variable* cannot be accessed by other functions in the program – it is in scope only for the code of its own function – but a value assigned to it will still exist the next time the function is entered. A static variable defined as a member of a class is in scope for all members of that class. There is only one copy of such a static member, no matter how many instances of the class are created – a static class data member is often referred to as a *class variable*. An external variable is, by default, of static storage class.

If a static variable is not explicitly initialised, its value is set to zero at compile time when space for it is allocated. Here is an example of how a static variable internal to a function retains its value even on exit from the function:

```
void run_total(void)
{
    static int total = 1;

    total = total + 1;
}
```

At compilation time, the compiler allocates space for total and sets its value to 1. This is done *only once*, not every time the function is entered. Every time the function is executed, the value of total is incremented by one. The fourth time that the function is entered, the value of total is 4.

The extern specifier

The last storage class specifier is extern. An external variable may be accessed by any function in the program file in which it is defined. Its definition may be accessed by any function in another program file if it is specified in that program file with the keyword extern.

Here is a trivial pair of program files that illustrate an extern declaration used in one to allow its code to access a global variable defined in the other. The files are extern1.cpp and extern2.cpp.

```
//   extern1.cpp
#include <iostream>
using namespace std;

void func1();   // prototype

int x = 5;   // global variable
int main()
{
    cout << "Value of x is: " << x <<endl;
    x = 7;
    func1();
    cout << "Value of x is: " << x <<endl;
}

//   extern2.cpp
#include <iostream>
using namespace std;

extern int x;          // external reference to global variable
void func1()
{
    cout << "Value of x is: " << x <<endl;
    x = 9;
}
```

The first action in the main function is to report the initialised value of the global variable x, which is then assigned the value 7. Then, the function func1 (in the second program file) is called to report the new value and again reassign x. Control is returned to main, where the final value of x is reported. From this, it is clear that the global variable is visible in both program files. For x to be in scope for any other program file that might be added, that program file must contain the same **extern** declaration as does **extern2.cpp** above. The displayed output of the program is this:

```
Value of x is: 5
Value of x is: 7
Value of x is: 9
```

Functions are not variables, but they are external objects: they are accessible throughout the whole program. The start of a function is the first character to the right of the opening parenthesis in front of the formal parameter declarations. All data objects declared within a function are internal. The function name is external because it is not part of the function itself. This in turn means that functions must not be nested: you can't define a C++ function within another function.

Putting it all together

At the start of this chapter, I refer to three kinds of scope: global (file), namespace and class. C++ has two additional scopes: function (only relevant with the goto statement; more in Chapter 6), and local (enclosing {} block). Here is an example program, stored in the program files progf1.cpp and progf2.cpp, that illustrates many aspects of global, namespace, class and local scope. First, a header file proghead.h, containing necessary class, namespace and function declarations:

```
namespace ns1
{
    void func1();
}

namespace ns2
{
    void func1();
}

class c
{
private:
    int x;
public:
    static int y;
    void func1();
};

void func1();   // prototype of global func1()
void func2();   // prototype of global func2()
void func3();   // prototype of global func3()
```

You can see that there are to be (quite deliberately!) four versions of the function func1. There is one in each of the namespaces ns1 and ns2, one in the class c and one in global scope. The program following shows how func1 is called in each case by indicating the scope of the version of the function that is required.

```
/*
 *  Program file (translation unit) progf1.cpp
 */
#include <iostream>
#include "proghead.h"

int x = 30; // global variable
int c::y = 50;   // class static variable

void c::func1()
```

```
{
    x = 20; // assign to class-instance variable
    std::cout << "In class func1()" << std::endl;
    std::cout << "Global x is " << ::x << " Class x is " << c::x
        << " Class static y is " << y << std::endl;
}

int main()
{
    c c_inst;
    c_inst.func1(); // call class func1
    func1();        // call global func1
    func2();
    c_inst.func1();
    func3();
    func3();
    ns1::func1();   // call func1 in first namespace
    ns2::func1();   // call func2 in second namespace
}

void func1()
{
    std::cout << "In global func1()" << std::endl;
}

void ns1::func1()
{
    std::cout << "In ns1::func1()" << std::endl;
}
```

This is the second program file, progf2.cpp:

```
/*
 *  Program file (translation unit) progf2.cpp
 */
#include <iostream>
#include "proghead.h"
extern int x;    // reference to global variable
void func2()
{
    int x = 10; // local (func2) variable
    std::cout << "In global func2()" << std::endl;
    ::x = 31;    // assign to global variable
    c::y = 51;   // assign to static class variable
}
```

```
void func3()
{
    static int y = 40;
    std::cout << "In global func3()" << std::endl;
    std::cout << "Value of local static is " << y << std::endl;
    y = 50;
}
void ns2::func1()
{
    std::cout << "In ns2::func1()" << std::endl;
}
```

This program is a bit complex but is very useful in that it displays in one place so many of the possibilities. Do your best to follow it: you'll find it worth it when handling real, complex, C++ programs.

As well as there being four versions of func1, there are three different declarations of the variable x. The first of these is the definition and initialisation of the global x at the top of progf1.cpp. The second is a member of the class c (declared in the header file). The third is a local (automatic storage class) definition of x, to be found in the global function func2 in program file progf2.cpp.

The main function creates an instance of the class c and calls its member function func1 to assign a value to c::x and report its value and the value of global x.

The static class-member variable c::y is initialised in global scope before the main function:

```
int c::y = 50; // class static variable
```

The function that now does most of the 'business' is global func2, called from main. It reassigns the global variable x and the class static c::y. When control returns to main, c::func1 is again called to report the new values of the global and class static variables. The global function func3 is then called twice to show the behaviour of the internal (to func3) static variable y. Finally, use of two namespaces, ns1 and ns2, shows how we can have two more instances of func2, neither in global nor class scope, and neither clashing with the other.

The program is compiled and linked with the following command-line sequence:

```
bcc32 progf1.cpp progf2.cpp
```

and run with the command

```
progf1
```

Its displayed output is this:

```
In class func1()
Global x is 30 Class x is 20 Class static y is 50
```

```
In global func1()
In global func2()
In class func1()
Global x is 31 Class x is 20 Class static y is 51
In global func3()
Value of local static is 40
In global func3()
Value of local static is 50
In ns1::func1()
In ns2::func1()
```

Declaration vs definition

The terms "declaration" and "definition" tend to be used as synonyms, which they are not. In C++ the difference is very important. A declaration is an announcement to the compiler that there is or will be a definition somewhere else in the program. A declaration does not have any memory space allocated to it. Different kinds of declarations include function prototypes, class declarations and extern specifications.

A definition is the actual object, not just an announcement of it. Every definition has memory allocated for it. Examples include the code (header and body) of a function as opposed to its prototype, a class instance, and the definition of a global variable, as opposed to a corresponding extern declaration.

Because declarations do not have memory allocated for them, they are suitable for inclusion in header files. It is possible, but not advisable, to put definitions in header files. A header file containing a definition will cause linker errors if it is included twice or more in the program files that make up the program as a whole.

Overloaded functions

C++ introduces overloaded functions. You can use a single function name to refer to more than one instance of the function, with each instance of the function having different argument lists. The compiler discriminates between function instances using the different argument lists in their prototypes.

Function overloading gives you flexibility. You can use the same function name to carry out operations on different data without having to be aware of how those operations are implemented. Suppose, for example, you want to find the product of two numbers, either of which may be of type int or double. You declare and define four functions, all with the same name, to ensure a correct result regardless of the types of the arguments used in a call to the function.

```
// Overloaded function prototypes

int prod_func(int, int);
double prod_func(int, double);
double prod_func(double, int);
double prod_func(double, double);
```

The full text of the functions is not shown; we assume that the types and arithmetic operations are properly handled by them. The C++ compiler chooses the appropriate function instance depending on the syntax of the function call that you write:

```
double prod;

prod = prod_func(15, 2.718281828);
```

This code causes the function declared by the second prototype to be called.

Example: overloaded functions

Here's a program, **fnovl1.cpp**, that uses overloaded functions to find the squares of numbers:

```
#include <iostream>
using namespace std;

// Function 'sqr_func' overloaded

float    sqr_func(float);
double    sqr_func(double);
double    sqr_func(float, float);

int main()
{
    float  f = 1.7320508;
    double d = 2.236068;

    cout << "Square of " << f << " is: " << sqr_func(f) << endl;
    cout << "Square of " << d << " is: " << sqr_func(d) << endl;
    cout << f << " multiplied by itself is " << sqr_func(f, f) << endl;
}
```

```
float sqr_func(float f)
{
    return(f * f);
}
double sqr_func(double d)
{
    return(d * d);
}
double sqr_func(float f1, float f2)
{
    return(f1 * f2);
}
```

The results output by this program are:

```
Square of 1.73205 is: 3
Square of 2.23607 is: 5
1.73205 multiplied by itself is 3
```

There are three instances of sqr_func, all with different argument lists. The compiler selects the appropriate function depending on the arguments used in the function call. The criteria the compiler uses to make the selection are explained in the next section. Some basic selection rules follow.

- The compiler does not use the return type of the function to distinguish between function instances.
- The argument lists of each of the function instances must be different.
- Whether or not argument names supplied on a function call match the corresponding parameter names in the function definition does not affect the selection process.

Use of prototypes such as these, with matching function definitions later in the code, results in compilation errors:

```
float sqr_func(float);
double sqr_func(float);
```

The compiler regards the function sqr_func as having been defined identically twice, regardless of the different return types.

Function call resolution

When you call an overloaded function, there are three possible results:

- A single function instance is matched by the compiler to the function call and this instance is called.
- Multiple, ambiguous, matches are found by the compiler, which is unable to select between them. A compilation error results.
- No match can be found by the compiler and a compilation error results.

This program, fnovl2.cpp, illustrates all three cases. Note that two of these cases cause compilation errors that are explained below.

```
#include <iostream>
using namespace std;
float sqr_func(float);
double sqr_func(double);
int main()
{
    float  f = 1.7320508;
    double d = 2.236068;
    int    i = 5;
    int    *ip = &i;
    cout << "Square of " << f << " is: "
             << sqr_func(f) << endl;
    cout << "Square of " << d << " is: "
             << sqr_func(d) << endl;
    cout << "Square of " << i << " is: "
             << sqr_func(i) << endl;
    cout << "Square of " << ip << " is: "
             << sqr_func(ip) << endl;
}
float sqr_func(float f)
{
    return(f * f);
}
double sqr_func(double d)
{
    return(d * d);
}
```

The calls to the function sqr_func using double and float arguments are successfully matched.

The call using the integer argument is matched by promotion of the integer to either float or double type but is ambiguous and causes a compilation error: the compiler does not know which function instance to call as either promotion is equally valid.

There is no matching function declaration or definition for the call using the pointer argument. This causes a compilation error.

When using overloaded functions, you should ensure that the order and type of arguments in the call match the argument list in one (and only one) instance of the overloaded function. If the match is not exact, the C++ compiler will try very hard to resolve the function call to a match, but it's better to avoid this altogether.

Function templates

C++ provides function templates so that you can define a function capable of operating on arguments of any type. You declare a function template by prefixing a function declaration with the template keyword followed by a pair of angle brackets containing one or more identifiers that represent *parameterised types*. This construct is called the *template specification*.

C++ is a strongly typed language. This is mostly a benefit, promoting program reliability, but it causes problems when you need to call a simple function with arguments of types that may vary from call to call. A good example is a function called min that must find the minimum of two values supplied as arguments. If the function on a first call is to compare two ints and on the second two doubles, then conventionally you have to make two definitions of the function to handle the two different calls.

Templates provide an elegant solution to this problem as you can see from the following example program.

```
#include <iostream>
using namespace std;

//  template declaration
template<class num>
num min(num n1, num n2);

int main()
{
    int i1, i2;
    double d1, d2;

    cout << "Enter two integers: ";
    cin >> i1 >> i2;
    cout << "minimum is: " << min(i1, i2) << endl;

    cout << "Enter two doubles: ";
    cin >> d1 >> d2;
    cout << "minimum is: " << min(d1, d2) << endl;
}

//  template definition
template<class num>
num min(num n1, num n2)
{
    if (n1 < n2)
        return (n1);
    return (n2);
}
```

In this program, we define a function template that expects one type parameter represented by the place-holder num specified between angle brackets following the template keyword. On the first call to min,

```
min(i1, i2)
```

an instance of the function template is created. This process is said to *instantiate a template function*. The resulting template function has the type of the two arguments, int, substituted for the placeholder num and compares two integers. On the second call to min

```
min(d1, d2)
```

a second template function is instantiated. This function has double substituted for num and compares two double floating-point numbers. The program's input/output sequence is this:

```
Enter two integers: 3 4
minimum is: 3
Enter two doubles: 3.5 4.5
minimum is: 3.5
```

You should be able to see that the compiler instantiates two template functions called min. In general, template functions are instantiated when the function is called or its address taken. The types of the arguments used in the function call determine which template function is instantiated:

```
min(i1, i2);
```

This causes a template function to be instantiated with the type parameter num becoming int.

Exercises

1. Write a program that, in its main function, repeatedly calls the function **run_total** with a single integer argument. The matching parameter in the header of **run_total** is called increment. Within the function, add increment to an accumulator and display the accumulator's value. That value should be the cumulative value for all the times **run_total** has so far been called.

2. Write a program that, in its main function, calls the function **get_num** with a single address of integer argument. **get_num** should read a number from the standard input. On return from **get_num**, display that number.

4 Aggregate types

Defining and initialising arrays

Definition

Here is a definition of an array of data objects of type int

```
int     numbers[10];
```

Ten integer data objects are defined. They are individually accessed by means of the identifier numbers suffixed by a subscript enclosed in square brackets

numbers[0] is the first (element zero), leftmost integer in the array
numbers[1] is the second.

numbers[9] is the last, or rightmost, element.

Subscripts in C++ always start at zero and stop one short of the subscript limit given in the array definition. Subscript values at the time of array definition must be constants. C++ does not allow variable-bound array definitions. You can define arrays of objects of any data type. Both the following are fine

```
char     charray[20];
float    flarray[50];
```

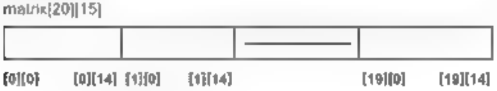
You can also define arrays of pointers and arrays of aggregate data types, including arrays, structures and other (programmer-defined) data objects.

Here is a definition of a multi-dimensional array – an array of arrays

```
int     matrix[20][15];
```

You can define arrays in any number of dimensions. If you find it easy to use arrays of four or more dimensions, then your intelligence is superior to mine. Considering the array matrix

- There are 20 rows, counted from zero to 19.
- There are 15 columns, counted from zero to 14.
- matrix[14][11] can be thought of as the element at row 14, column 11
- matrix[14][11] is more accurately thought of as the 11th element of array 14
- The array matrix is not, in fact, organised in memory as a rectangle of integer data objects; it is a contiguous line of integers treated as 20 sets of 15 elements each
- The subscripts of matrix are specified in row-column order – matrix[r][c] – and the column subscript varies more rapidly than the row subscript, in line with the way in which the array elements are stored in memory



Initialisation

You can explicitly initialise an array using an *initialiser list* consisting entirely of constant values. Any initial values of automatic array elements that are not explicitly initialised are garbage; for static arrays, the array elements are zero.

You can enclose array initialiser lists in curly braces, as in the example below, or in the case of string initialisation, use a string literal enclosed by double quotes.

You could define and initialise an array mdd to hold the number of days in each month of the year:

```
int mdd[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

The data used to initialise an array must be a set of constants enclosed in curly braces, separated by commas and terminated with a semicolon.

In the same way, you could set up a character array:

```
char arr[5] = {'h','e','l','l','o'};
```

If there are more initialising data objects within the curly braces than implied by the subscript limit, the compiler reports an error. If there are fewer initialising data objects than the subscript limit, all excess elements in the array are set to zero for a static array, but have garbage values for an automatic array.

You initialise two- and multi-dimensional arrays like this:

```
int tab[3][4] = {
                    {1,2,3,4},
                    {5,6,7,8},
                    {9,10,11,12}
                };
```

Here the outside curly braces are necessary, while the internal ones are optional but are included for readability. The first array bound, [3], is the number of rows and the second, [4], represents the number of columns. The array tab consists of three four-element integer arrays. The contents of some of its elements are:

```
tab[0][0] == 1
tab[1][2] == 7
tab[2][3] == 12
```

You can leave out one of the array bounds, but not both:

```
int tab[][4] = {
                   {1,2,3,4},
                   {5,6,7,8},
                   {9,10,11,12}
               };
```

Strings, subscripts and pointers

Definition of C-string

You've already seen (in Chapter 2) the difference between C-strings and instances of the C++ Standard Library class string. This section is concerned with C-strings. C-strings are arrays of elements of type char terminated by the first null character '\0' encountered in the array. The definition from the last section

```
char  arr[5] = {'h','e','l','l','o'};
```

is a character array: elements 0 to 4 of the array are initialised with the five letters of "hello". The definition

```
char arr[6] = {'h','e','l','l','o','\0'};
```

is a C-string initialised to "hello" and null-terminated. To accommodate the null character, the second definition of arr needs one more element.

The last definition is equivalent to:

```
char arr[6] = "hello";
```

Recall that a character constant, which can only represent one character, is delimited by single quotes as in 'a' and '\n'. A string literal is delimited by double quotes. The double quotes imply the existence of a terminating null character. If you forget to add one more array element to accommodate the null character, ISO C++ does not add one for you and 'get you out of trouble'. However, your C++ compiler will probably do it for you, as shown by the program null.cpp

```
#include <iostream>
using namespace std;

int main()
{
   char arr[5] = "hello";

   for (int i=0; i<10; i++)
      if (arr[i] == '\0')
         cout << " Null ";
      else
         cout << arr[i] << " ";
}
```

The displayed output on my system is

```
h e l l o  Null  Null  Null , ÿ
```

which shows that space was allocated for at least one '\0' character following the five characters of "hello".

The single-quoted sequence 'a' is the mechanism used in C++ to represent the ASCII code table entry for the letter a. By contrast, "a" is a null terminated C-string, equivalent to the character pair 'a','\0'.

You can use string literals in the same way as variable C-strings, and even access individual characters of a string literal by suffixing the literal with a subscript:

```
"hello"[1] == 'e'
```

Each time you use a string literal in a program, the compiler may allocate new memory space for it – an unnamed static array – even if it is identical to a string literal used earlier in the program. By contrast, a variable array, once defined either externally or internally, will only have one memory allocation made for it at any given time. You shouldn't use an error message such as

```
"Error: can't open file"
```

repeatedly in a program in its string-literal form. Instead, it's best to use the literal to initialise a variable and then to use the variable with successive printf calls.

Finding string length using subscripts

The program that follows, slengths.cpp, calculates the length (counting from 1, not zero) of a C-string. The call to slength – which traverses the C-string to find its length – is included as part of the second cout statement in main. Every function has a return value and type; slength is of type int, so the call to it is treated as an int in the cout statement.

```
/*********************************************************************
 *     'slengths.cpp'     Find the length of a C-string stored in
 *                 a character array, using subscripts
 *********************************************************************/
#include <iostream>
using namespace std;

int slength(char []);

int main()
{
      char instring[50];

      cout << "Enter input string: ";
      cin >> instring;
      cout << "String length is " << slength(instring) << endl;
}

int slength(char instring[])
{
      int i;

      for (i=0; instring[i] != '\0'; i++)
            ;
      return(i);
}
```

slength traverses the array until a null character is encountered, counting the number of characters on the way. The body of the for loop doing this consists of just a semicolon, which is a *null statement*.

The counting of characters includes the null character '\0'. Normally, the null character is not included in a C-string's length. However, we want to report the C-string's length as if we were counting from 1, not zero. Counting in the null character compensates for the fact that we are counting from zero. Here's the displayed output when the program is run:

```
Enter input string abcdefghijkl
String length is 12
```

If the C-string is not terminated with a null character, **slength** will run on until it reaches the end of the system's memory or it is stopped by the operating system. C++ does not check for this and all kinds of ghastly errors can result from omitting C-string terminators. In fact, one of the main motivations for providing a fully-functional string class as part of the C++ Standard Library is to make it unnecessary for you to use C-strings and run the risks of omitting the null terminator.

Finding string length using pointers

Given these definitions:

```
char     stg[50] = "Double double toil and trouble";
char     *cptr, *lptr;
```

and initialising the pointers,

```
cptr = lptr = stg;
```

the code following finds the length of the C-string as it has been initialised:

```
while (*lptr)
    lptr++;
return(lptr - cptr);
```

This is part of the pointer version of the **slength** function, which is given in full below. Note that:

- **cptr** and **lptr** point at the address of the first element of the character array **stg**.
- ***lptr** is the contents of that element.
- ***lptr** is the same as **stg[0]**.
- **lptr** is the same as **&stg[0]**.

When lptr is incremented by one, ***lptr** is equivalent to **stg[1]**; when further incremented by one, ***lptr** is the same as **stg[2]**, and so on.

lptr is incremented until its contents (*lptr) equal the null character. If *lptr is null, it is inherently false, so you don't have to make an explicit comparison between it and '\0'. The displacement of the pointer lptr from the array address stg (same as cptr) is calculated by subtraction giving the length of the C-string stg.

Here is the pointer version of the C-string-length program slengthp.cpp.

```
/**********************************************************************
 *
 *     'slengthp.cpp'     Find the length of a C-string stored in
 *                  a character array, using pointers
 *
 **********************************************************************/
#include <iostream>
using namespace std;

int slength(char *);

int main()
{
      char instring[50];
      char *cptr = instring;

      cout << "Enter input string ";
      cin >> cptr;
      cout << "String length is " << slength(cptr) << endl;
}

int slength(char *cptr)
{
      char *lptr = cptr;

      while (*lptr)
            lptr++;
      return(lptr - cptr);
}
```

The integer number returned by slength to main is the displacement between the two pointers lptr and cptr and is the length of the C-string instring. Executing this program produces the same output for given input as does slengths.cpp.

C library string functions

Common operations on strings include copying, length checking and comparison. Using the standard header file cstring, you can use the traditional C library functions for string handling. (As part of the ISO C++ standardisation, the then Standard C++ header file string.h was renamed cstring to make clear the purpose of the file. The header file string is what you now include when you want to use the facilities of the Standard C++ string class. All other traditional C header files have been similarly renamed: stdio.h to cstdio, stdlib.h to cstdlib and so on.) To use cstring, you should include it in your program using the preprocessor

```
#include <cstring>
```

Some of the most often-used C-string functions are

strlen	Finds the length of a string
strcat	Joins two strings
strcpy	Copies one string to another
strcmp	Compares two strings
strncmp	Compares parts of two strings

strlen operates like slengths and slengthp from the previous section

```
int len;
char  s[50] = "A text string";

len = strlen(s);
```

After this code len contains the number of characters in the C string (13), not counting the terminating null character.

strcat concatenates two C-strings:

```
char s1[50] = "A text string ";
char s2[50] = "with another appended";

strcat(s1, s2);
```

This appends the C-string s2 to the end of the C-string s1, yielding "A text string with another appended" as the contents of s1. It's your responsibility to ensure that s1 is long enough to accommodate the joined C-strings. In real application programs, the first argument to strcat is usually a pointer pointing to enough dynamically allocated memory to ensure that both C-strings can be accommodated. *Dynamic memory allocation* is explained in Chapter 7.

strcpy copies the second C-string operand to the first, stopping after the null character has been copied. The example given below illustrates its use. strcmp and strncmp both compare two C-strings and return a negative, zero or positive value, depending on whether the first string is lexicographically less than, equal to or greater than the second.

```
char s1[50], s2[50];
```

```
strcpy(s1, "hello");
strcpy(s2, "hallo");

result = strcmp(s1, s2);

// s1 greater than s2, so result is positive
```

strncmp does the same thing as strcmp, but only compares a specified number of characters in the two C-strings:

```
strncmp(s1, s2, 1);
```

In the example above, this would compare only the first letters of the two strings and would return a zero value denoting equality. You're not allowed in C++ to compare two strings using the == equality operator unless you use operator overloading to give the == operator a new definition allowing it to do that comparison. For more on operator overloading, see Chapter 9. Assuming the non-overloaded ==, each character in the two C-strings must be compared to its counterpart in the other C-string. The library functions strcmp and strncmp are provided for this purpose.

Program example: pattern matching

The program strpos.cpp that follows accepts as input two strings s1 and s2 and finds the start position in s1 of s2. If s2 is not found in s1, a negative value is returned. The strpos function, which finds the position of s2 in s1 if there is a match, is an extremely useful function for pattern matching in text. Its C library equivalent is strstr.

```
/************************************************************
 *
 *     'strpos.cpp' — Find the position of C-string s2
 *            in s1. Return the position if found,
 *            or a negative value otherwise
 *
 ************************************************************/
#include <iostream>
using namespace std;

#include <cstring>
#define MAX 50

int strpos(char *, char *);

int main()
{
    char    str1[MAX], str2[MAX];
    char    *s1 = str1, *s2 = str2;
    int     pos;

    cout << "Enter string to be searched: ";
    cin.get(s1, MAX);
    cin.get();              // Read trailing '\n'
    cout << "Enter search string: ";
    cin.get(s2, MAX);
    cin.get();              // Read trailing '\n'
    pos = strpos(s1, s2);
    if (pos < 0)
        cout << s2 << " not found in " << s1 << endl;
    else
        cout << s2 << " at position "
             << pos << " in " << s1 << endl;
}

int strpos(char *s1, char *s2)
{
    int     len;
    char    *lptr = s1;

    len  = strlen(s2);

    while (*lptr)
    {
        if ((strncmp(lptr, s2, len)) == 0)
            return(lptr - s1 + 1);
        lptr++;
    }
    return(-1);
}
```

The main function accepts user input of two C-strings and then passes them to strpos for matching. The statements

```
cin.get(s1, MAX);
cin.get();
```

are used as an alternative to:

```
    cin >> s1
```

which is what we have used for input up to now. The difference between the two is that the function cin.get will accept input text containing blanks, while the simple use of cin stops input to s1 when it encounters the first blank.

Let's use an example to explain how the function **strpos** works. Suppose **s1** points to the C-string "Great Dunsinane he strongly fortifies" (more Shakespeare), while **s2** points to another C-string "Dunsinane". The first thing the function does is to set a temporary pointer **tptr** equal to **s1** and thus pointing at the longer C-string. A call to **strlen** finds the length of the C-string at **s2** which, in the case of "Dunsinane", is 9. Inside the loop, while **tptr** is still pointing at a non-null character, **strncmp** is used to compare "Dunsinane" with successive nine-character substrings of the longer C-string. If there's a match, **strncmp** returns zero and our **strpos** function returns the position of "Dunsinane", that is, 7. If no match is ever found, **strpos** returns the position as -1.

Here's the display produced by **strpos.cpp**:

```
Enter string to be searched: Great Dunsinane he strongly fortifies
Enter search string: Dunsinane
Dunsinane at position 7 in Great Dunsinane he strongly fortifies
```

Structures

The elements of an array are all the same size and type. If you need to group together in one entity data objects of different sizes and types, you can use structures to do so. A structure is an aggregate data type: a collection of variables referenced under one name. A structure declaration is a *programmer-defined data type*. The C++ class construct (introduced in Chapter 1, covered more fully in Chapter 8) is simply a special kind of structure. The only difference is that, with structures, the default access level for members is public; access to class members is by default private. Structure members can be either data or functions.

Structure declaration and instantiation

Here is how to declare a structure:

```
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;

    int       stock_take();
    void      reorder(int);
    void      take_from_stock(char *);
    void      show_values();
};
```

Notice that there are six data members and four function members. Data and function members could be interspersed in any order; it does not have to be all data followed by all functions. Access to all ten members is set by default to public; any outside function can call the member functions or directly use the data members. From the standpoint of object-oriented design good practice, it is best not to have everything public, which is why C++ classes are used much more than structures. There remains a place in C++ for structures for cases in which a collection of data of different types needs to be stored in a single entity and where data-hiding is not a major consideration. This is usually found in a class library where the classes need free access to a collection of data; the collection is represented as a structure and used internally only by the classes.

The structure declaration above is not a definition – no memory space is allocated for the data objects specified, and it does not create an instance of (instantiate) the structure. All that exists after the declaration is the new, programmer-defined data type stock_type. This is a grouping of data and function declarations that may be used to instantiate, or define, structure variables. To define a structure variable with a *variable list*, you can use this form:

```
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;

    int       stock_take();
    void      reorder(int);
    void      take_from_stock(char *);
    void      show_values();
}stock_item;
```

Now, we have defined an instance of the data type stock_type, for which memory is allocated and which is called stock_item. You can put multiple names in the variable list to define multiple instances of the structure.

There is a better way of defining instances of a structure, e.g.:

```
stock_type stock_item1;
```

creates an instance of the stock_type structure in memory and separates the declaration of the structure from its definition. This method allows the programmer to put the structure declaration in a #include file and later to define instances of that declaration in the program.

Structure members

The component data and functions of a structure are called members. In the stock_type example, there are ten members of the structure and every instance of the structure has the same ten members. To refer to an individual structure member you use this syntax:

```
stock_item1.reorder(100);       // order 100 more
```

The 'dot' or 'member of' operator references reorder as a member function of stock_item1, which is defined as an instance of the structure type stock_type.

There's nothing wrong with defining an array as a member of a structure. You access the fifth element of the array item_name like this:

```
stock_item1.item_name[4]
```

A structure may have one or more members which are also structures. A structure must not contain an instance of itself.

It's legal to assign to a structure another structure of identical type. However, you can't compare two structures using the default (non-overloaded) equality operator ==. Each of the structure members must be individually compared.

```
stock_item2 = stock_item1;          // assignment, OK
if (stock_item1 == stock_item2)     // comparison, wrong
```

Nested structures

You can define a structure member that is itself a structure. Here's an example of nested structure declarations and definitions.

```
struct stock_type
{
   char  item_name[30];
   char  part_number[10];
struct detail
{
   int      height;
   int      width;
   int      depth;
   struct bin
   {
      char      building[50];
      int      floor;
      int      bay;
      int      shelf;
      int      quantity;
   }bin_loc;
   char        special_reqs[50];
   char        part_number[10];
}item_detail;
   double   cost_price;
   double   sell_price;
   int         stock_on_hand;
   int         reorder_level;

   int         stock_take();
   void        reorder(int);
   void        take_from_stock(char *);
   void        show_values();
};

// define an instance of the outermost structure
stock_type stock_item;
```

The structure item_detail is nested within stock_item and contains further information about the stock item. The structure bin_loc is in turn nested within item_detail and holds information about a bin location. In C++, it is unusual to nest structures fully in this way. The more typical (and equivalent) syntax is this:

```
struct bin
{
    char      building[50];
    int       floor;
    int       bay;
    int       shelf;
    int       quantity;
};
struct detail
{
    int       height;
    int       width;
    int       depth;
    bin       bin_loc;
    char      special_reqs[50];
    char      part_number[10];
};
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    detail    item_detail;
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;

    int       stock_take();
    void      reorder(int);
    void      take_from_stock(char *);
    void      show_values();
};
stock_type stock_item;
```

In either case, you find the height of a particular item with this code:

```
stock_item.item_detail.height
```

and the shelf on which that item is stored is:

```
stock_item.item_detail.bin_loc.shelf
```

In the second form of declaration, the three structures are declared and defined in reverse order. You have to do this to conform with C++'s scope rules for declarations of variables. The declaration of detail is in scope for the definition of item_detail because it appears first. If the declaration of detail were instead to follow that of stock_type, a compiler error would result, flagging detail as an unknown type.

You can use the name of a structure member either in other structures or as the identifier for an elementary data object without any clash. In the example above, uniqueness of the identifier part_number is ensured by the fact that it must be suffixed to a structure name by the dot operator:

stock_item.part_number

Using structure instances

Let's look at a simple program, initstr1.cpp, which assigns values to the members of a structure of type stock_type and displays the contents.

```
/*********************************************************************
 *
 *     initstr1.cpp — Creates a structure instance and assigns data
 *                    to its data members. Then calls member function
 *                    show_values to display those values.
 *
 *********************************************************************/
#include <iostream>
using namespace std;

struct stock_type
{
    char        item_name[30];
    char        part_number[30];
    double      cost_price;
    double      sell_price;
    int         stock_on_hand;
    int         reorder_level;
    int         stock_take();
    void        reorder(int);
    void        take_from_stock(char *);
    void        show_values();
};

int stock_type::stock_take()
{
    // do stock-taking
    return 0;
}

void stock_type::reorder(int reorder_qty)
{
    // reorder the quantity
}
```

```
void stock_type::take_from_stock(char *part_no)
{
    // take the part from stock
}

int main()
{
    stock_type stock_item;

    const int MAX = 50;

    cout << "Enter item name ";
    cin.get(stock_item.item_name, MAX);
    cin.get();
    cout << "Enter part number ";
    cin >> stock_item.part_number;
    cout << "Enter cost price ";
    cin >> stock_item.cost_price;
    cout << "Enter sell price ";
    cin >> stock_item.sell_price;
    cout << "Enter stock on hand ";
    cin >> stock_item.stock_on_hand;
    cout << "Enter reorder level ";
    cin >> stock_item.reorder_level;

    stock_item.show_values();
}

void stock_type::show_values()
{
    cout << endl << item_name << endl;
    cout << part_number << endl;
    cout << cost_price << endl;
    cout << sell_price << endl;
    cout << stock_on_hand << endl;
    cout << reorder_level << endl;
}
```

The declaration of the stock_type class specifies six data and four function members, all with public access. Three of the four member functions are defined as dummies, for example

```
int stock_type::stock_take()
{
    // do stock-taking
    return 0;
}
```

This is the header and body of the function stock_take which is in the scope of the structure stock_type and returns a value of type int.

The main function first defines an instance of the structure type stock_type. For all the members of the structure, cout statements prompt the user to enter data. Depending on whether or not expected data input will contain blanks, either cin.get or cin is used to read input directly into the data members of stock_item. Finally, the member function show_values is called to display the contents of the structure members.

Defining an array of structures

You can define arrays of structures in the same way as arrays of any other data object. Look at the structure struct bin:

```
struct bin
{
    char   building[50];
    int    floor;
    int    bay;
    int    shelf;
    int    quantity;
}
```

There are probably many bin locations where a given item is stored, perhaps dispersed among different buildings. Each bin location might be numbered up to a maximum. You can hold all the bin location detail in an array of structures of type bin:

```
bin bin_arr[20];
```

Now you can search for a bin which has one of the items in stock:

```
for (int i=0; i < 20; i++)
{
    if (bin_arr[i].quantity != 0)
    {
        // Item found
        cout << "bay " << bin_arr[i].bay
            << " shelf " << bin_arr[i].shelf
            << " in building " << bin_arr.building << endl;

        // Take one out of stock
        bin_arr[i].quantity -= 1;
        break;
    }
}
```

Initialising a structure instance

You will remember that initialisation of a variable happens at the point of its definition, while assignment takes place sometime after the definition. The program initstr.cpp makes assignments to a structure instance. Now we're going to see how to initialise one. In fact, initialising a structure is like initialising an array.

Using the familiar declaration and definition of stock_type and stock_item, here is how stock_item is initialised:

```
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;

    int       stock_take();
    void      reorder(int);
    void      take_from_stock(char *);
    void      show_values();
};

stock_type stock_item =
{
    "Turbocharged sewing machine",
    "8705145B",
    275.65,
    340.00,
    50,
    20
};
```

All the initialising expressions should be of the same types as the corresponding structure members, otherwise these will end up containing corrupted data. Similarly, the initialising string constants should be shorter than the sizes of the array members of the structure to allow inclusion of the null character as terminator.

Using the mutable keyword

The storage class type qualifiers const, volatile and mutable were introduced in Chapter 2. Now that we've seen the C++ struct, I can explain the use of mutable. When creating an instance of a structure or a class, you can qualify the definition with const so that the instance members cannot subsequently be changed. But you might want a situation where not all the members of the structure or class are to be const and unchangeable after the definition. For example, in the initialised structure instance we have just seen, it might be necessary to change the selling price, even though everything else remains immutable. Hence mutable.

```
#include <iostream>
using namespace std;

int main()
{
    struct stock_type
    {
        char        item_name[30];
        char        part_number[30];
        double      cost_price;
        mutable     double      sell_price;
        int         stock_on_hand;
        int         reorder_level;

        int         stock_take();
        void        reorder(int);
        void        take_from_stock(char *);
        void        show_values();
    };

    const stock_type fixed_item =
    {
        "Turbocharged sewing machine",
        "8705145B",
        275.65,
        340.00,
        50,
        20
    };

    fixed_item.sell_price = 400.00; // OK
}
```

Pointers to structures

Pointers to structure members

You can define a structure and a structure pointer like this:

```
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;

    int       stock_take();
    void      reorder(int);
    void      take_from_stock(char *);
    void      show_values();
};

stock_type stock_item;
stock_type *sptr = &stock_item;
```

Notice that sptr, the structure pointer, is initialised to the address of the structure instance stock_item. You can use the pointer sptr rather than the instance name stock_item to access the structure's members with the arrow operator:

```
sptr->part_number

sptr->part_number[5]

sptr->stock_on_hand

sptr->reorder(100)
```

Recall that sptr is pointing at the structure instance and that the 'object at' sptr (*sptr) is the structure data itself. Therefore, the syntax sptr-><member> is equivalent to (*sptr).<member>

Structures as arguments

You can pass structure instances as arguments between functions. It is more efficient to pass pointers to instances than the instances themselves:

```
void some_func(stock_type *);  // function prototype


stock_type stock_item;
stock_type *sptr = &stock_item
    some_func(sptr);                    // function call
```

```
void some_func(stock_type *ptrIn)    // function definition
{
    ptrIn->reorder(100);             // reorder stock
}
```

In modern C++, the reference type is preferred to pointers for use in passing structure and class arguments between functions.

```
void some_func(stock_type &); // function prototype


stock_type stock_item;

some_func(sptr);                     // function call



void some_func(stock_type& refIn)    // function definition
{
    refIn.reorder(100);              // reorder stock
}
```

Using references removes the need for use of a pointer address as the argument in the function call, and of the pointer-to operator -> in the called function. Some people like the symmetry and consistency of the pointer and dereferencing; newer C++ programmers tend to go with references. In any case, the compiler may internally implement the reference notation with the pointer equivalent. You should feel free to choose either option, although references are probably more fashionable.

It is almost always more efficient to pass large data objects, such as arrays, structures and class instances, as arguments between functions using their addresses, or references, rather than copying the whole structure. Copying structures between functions can result in significant overhead as member data of the structure is repeatedly pushed and popped on the system's stack space, which is used for transfer of arguments.

Unions

The C++ union is a special case of the class construct. A union is syntactically similar to a class (and a structure) but the compiler only allocates space in memory sufficient to accommodate the largest member of the union, along with any additional space at the end needed by the alignment requirements of the host computer system. At any given time, an instance of only one union member actually exists. You will only want to use unions in very specific cases, typically where memory space is at a premium.

Like classes, unions can have data and function members. They can also include constructor and destructor functions. (These are dealt with in Chapter 9). A union must not contain base classes or be itself a base class. Unions must not contain members that are virtual functions. Base and derived classes, along with virtual functions, are covered in Chapter 10.

It's OK for unions to have members specified private, protected or public. If none of these is specified, access defaults to public, in the same way as for a structure. Unions may be nested and may occur in arrays. Pointers to unions may be defined and the pointers or the unions themselves may be used as function arguments or return values.

Union example: a spreadsheet cell

Here's a complete example program that illustrates use of a union. The program accepts input to a character array and parses the contents as might an elementary spreadsheet program. The array is analysed to determine if its contents represent an integer, a double floating point number or a character string.

Depending on the type of the data, it is copied to the appropriate member of the union, displayed and then 'stored'.

```
#include <iostream>
using namespace std;

#include <cstdlib>      // declares 'atoi' and 'atof' functions
#include <cstring>      // declares C-string functions

//   Declare a union, it could be a simple spreadsheet cell

union sp_cell
{
private:
     int          ival;
     double       dval;
     char         sval[20];
     char         instring[20];  // array for input
```

```
public:
    char        get_token();
    char        analyse();
    void        put_token(char);
};

char sp_cell::get_token()
{
    cout << "Enter a number, fraction or string: ";
    cin  >> instring;
    return(analyse());
}

char sp_cell::analyse()
{
    int i = 0;

    while ((instring[i] != '\0') && (i < 20))
    {
        if (instring[i] == '.')             // decimal point
            return ('d');
        if ((instring[i] < '0') ||
                (instring[i] > '9'))
            return ('s');
        i++;
    }
    return ('i');
}

void sp_cell::put_token(char token_type)
{
    if (token_type == 'i')
    {
        ival = atoi(instring);
        cout << "Integer " << ival << endl;
    }
    else
    if (token_type == 'd')
    {
        dval = atof(instring);
        cout << "Double " << dval << endl;
    }
    else
    if (token_type == 's')
```

```
        {
            strcpy(sval, instring);
            cout << "String " << sval << endl;
        }
        else
            cout << "Invalid data" << endl;
        cout << "Data has been stored\n";
    }

    int main()
    {
        sp_cell cell;

        cell.put_token(cell.get_token());
    }
```

This program is an example of the remarkable brevity that can be achieved in the calling function by use of classes with encapsulated data and function members.

The union sp_cell is declared with three data members representing a spreadsheet cell: an integer, a double and a character array. A further character array, instring, is defined to take input data. The reason that we're using a union, as opposed to a struct or a class, is that the data input can be only one of the three possible types: whole-number, fraction (with a decimal point) or text string. There is therefore no need ever to allocate memory for a given spreadsheet cell for more than one of the three types. This makes the union ideal for this case.

The main function defines an instance, cell, of the union sp_cell. This is used to call the member function put_token. The call to put_token uses as its argument the return value of get_token, which prompts the user for input and accepts it into instring.

get_token in turn calls the function analyse, which does the string parsing and returns to get_token a character representing the type of data input. get_token returns the same character, which is used as a parameter by put_token to determine the nature of conversion and storage required by the input data.

Exercises

1. Write a program, **scopy.cpp** (without use of the library functions), that copies the contents of one C-string to another and displays the result.

2. Now write a cleverer program, **mincopy.c** (without use of the library functions), that uses the minimum amount of code necessary to do the work of **scopy.cpp**. (I give prizes for the shortest version of this program, so let me know if you think you've got a winner. The current champion has one (only) line of code in **main**.)

3. Write a program that accepts an input C-string. The contents of the C-string should be a sequence of characters in the range 0-9. There should be no more than six such characters. Validate the contents of the C-string as being an integer in the range 0-999999. Convert it to integer and display it. Use the library functions here if you need them.

4. Write a program that accepts an input C-string and then displays it with the characters in reverse order.

5 Expressions and operators

Boolean value of expressions

Every C++ expression has an inherent value, which is either zero or non zero. In the philosophically simple world of C++, everything is either true or false. Zero is "false", non zero is "true". As you've seen, C++ has a specific boolean data type bool; a variable of this type can have only two values, either true or false. But you can also use, for example, an int or short type if you want a boolean variable.

```
#define   TRUE    1
#define   FALSE   0
short  date_valid = 0;      // set FALSE

//  Assign return-value of date-validation function to the flag
//  The return value is either 0 (FALSE) or 1 (TRUE)
date_valid = validate();
```

Recall that every expression has a zero or non zero value.

```
if (date_valid) cout << "Valid date entered" << endl;
if (!date_valid) cout << "Invalid date entered" << endl;
```

If date_valid is not zero, it is "true" and the first test succeeds. If date_valid is zero, it is "false". The *unary negation operator* ! causes the second test to go true and an invalid date is flagged.

In this example, the function call validate() is itself an expression with an inherent return value. It's OK to write:

```
if ((validate()) == TRUE)          // return value true, date valid
    cout << "Valid date entered" << endl;
```

or simply

```
if (validate())cout << "Valid date entered" << endl;
```

For expressions, value zero represents "false", non-zero is "true". For relational expressions, "false" equals zero and "true" equals 1. To illustrate:

```
int     a = 0;
int     b = -5;
int     c = 5;
float   e = 2.71828
a          //  FALSE
b          //  TRUE
a + b      //  TRUE
b + c      //  FALSE
e          //  TRUE
a == 0         //  TRUE (1)
b < 0          //  TRUE (1)
e > 3.0        //  FALSE (0)
```

Finally, the explicit values 1 and 0 may be used to represent "true" and "false":

```
while (1)     //  infinite loop
while (!0)    //  same
```

Assignment

The simple assignment operator, =, has the effect of changing the value of the operand to its left. This operand is sometimes called the *lvalue*. The lvalue operand to the left of the assignment must be an expression referring to a region of memory which the program may change. Therefore it must not be a constant or an expression like x + 5.

```
 int no_leaps;

no_leaps = 0;   // assignment changes the value stored at the
                // location associated with the name no_leaps to zero
```

To add 1 to the value of no_leaps, you can use the traditional form of assignment:

```
no_leaps = no_leaps + 1;
```

This means that the memory location named no_leaps is updated with the current value stored in that memory location plus one. In C++ it is more common (and better practice) to use for the assignment that most characteristic of all aspects of C++ syntax:

```
no_leaps++;
```

This also increments no_leaps by one. Depending on the compiler, use of the post-increment may lessen compile time and reduce resultant code output, because there is only one reference to the variable being incremented.

Similarly, you can also write the decrement-by-one operation as:

```
no_leaps--;
```

To increment no_leaps by 2, you write:

```
no_leaps = no_leaps + 2;
```

or

```
no_leaps += 2;
```

This form of compound assignment can be generally applied:

```
x += y       is equivalent to    x = x + y
x *= y + z   is equivalent to    x = x * (y + z)
x -= y       is equivalent to    x = x - y
x /= y       is equivalent to    x = x / y
```

The most common of all compound assignments is the increment-by-one:

```
no_leaps++;
```

You can also use:

```
++no_leaps;
```

If no_leaps is the only operand in the expression and this expression is not itself on the right-hand side of an assignment, the two statements above are equivalent.

In the following case, the uses of the ++ compound operator before and after the variable are not equivalent

```
int days_total;

no_leaps = 5;
days_total = no_leaps++;
```

Here, the value of no_leaps is assigned to days_total and only then is no_leaps incremented by one. The value of days_total after the assignment is 5. If you instead wrote the assignment

```
days_total = ++no_leaps;
```

the value of no_leaps would be incremented first and then assigned to days_total giving a final value of 6.

Comparing data

Relational operators

To compare the values of variables in your C++ programs, you need relational operators. The relational operators provided by C++ are

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

The equality operators are

==	equality
!=	non-equality

All arithmetic operators are done before(have higher *precedence* than) relational tests, which in turn are carried out before tests for equality. For example

```
if (dd > MAXDD - 1)
```

means the same as

```
if (dd > (MAXDD - 1))
```

although you may often use the second form for clarity. For all operators, liberal use of parentheses can eliminate surprises caused by unexpected effects of the precedence rules.

Logical operators

C++'s logical operators are

&&	AND
||	OR
!	NOT

The precedence of the unary negation operator ! is the same as that of unary minus -, and is higher than any of the arithmetic, relational, or logical operators.

&& and || operations are of lower precedence than relational and equality operations. && is evaluated before ||. For example, evaluation of this *compound condition* will be unexpected:

```
if (mm==4 || mm==6 || mm==9 || mm==11 && dd>30)
```

The first test that is done is for mm being equal to 11 AND dd being more than 30. If the month is one of 4, 6 or 9, the test returns TRUE (1) regardless of the value of dd.

To achieve what was probably required – if it's April, June, September or November AND the day is greater than 30 – you should use parentheses that make clear the precedence you want:

```
if ((mm==4 || mm==6 || mm==9 || mm==11) && (dd > 30))
```

In general, if you are in doubt about default precedence, you should explicitly use parentheses to indicate what you intend. Even if they are unnecessary, they will be 'stripped out' at compilation time, and cost nothing in terms of the execution efficiency of your program.

Conditional expressions

The conditional operator ?: is the only so-called *ternary* operator implemented by the C++ language. The others are all either unary – in that the operator takes only one operand, or binary, taking two.

Using three operands, the ?: allows a shorthand to be used for if ... else constructs such as:

```
if (x > y)
    max = x;
else
    max = y;
```

Using ?: you can write this as:

```
max = (x > y) ? x : y;
```

The parentheses around the condition are not necessary: the ?: is of lower precedence than any of the arithmetic or logical operators.

Whether or not the condition expression is enclosed in parentheses, it is evaluated first; one – and only one – of the second and third operands is evaluated, depending on the boolean result of the condition expression.

The parentheses, even in this simple case, are nevertheless useful for readability.

One of the major uses of the ?: operator is for defining *preprocessor macros*. The ?: allows macros to be defined on one line and may cause the compiler to generate more efficient code than the if...else equivalent:

```
#define MAX(A, B) (A > B) ? A : B
```

After you make this preprocessor definition, all subsequent uses of MAX (for example MAX(5, 6)) are substituted in your program's code with the expression:

```
(A > B) ? A : B
```

which, in the example, evaluates to:

```
(5 > 6) ? 5 : 6
```

and eventually, 6.

Precedence and associativity

C's rules of precedence and associativity determine the order in which the operations making up the evaluation of an expression will take place. From the conventions of simple arithmetic, you would expect a * b + c to be evaluated as (a * b) + c and not a * (b + c). Some conventions hold that division is of higher precedence than multiplication, but in C++ they are the same, along with the modulus operator %.

Addition and subtraction are of the same precedence relative to each other, but lower than the other arithmetic operators.

Associativity is subordinate to precedence: when two operators are of the same precedence, the order of evaluation of the expression is controlled by their associativity. The expression following the definitions below uses all of the multiplicative operators, which are of equal precedence and associate left-to-right:

```
int a = 10;
int b = 5;
int c = 9;
int d = 4;

a / b * c%d  // 10/5*9%4  result is 2
```

You can now examine an almost-complete table of operator precedence and associativity. A few operators are included that haven't been encountered up to now. These are mainly concerned with advanced use of pointers, where precedence of pointer operators becomes important.

Unary -, + and * are of higher precedence than the same operators used with binary operands. The () operator means the parentheses in a function call. The [] operator means array-bound square brackets. Operators -> and . are the pointer-to and member-of operators for classes, structures and unions. The last operator in the table is the *comma operator*. This is infrequently used. When it is used, it separates two expressions, guaranteeing that the second expression is evaluated after the first.

A nod to the purists: there is some simplification of the comprehensive operator-precedence table. For example, distinctions are not drawn between the two uses of the scope resolution operator: that of specifying scope on the one hand and indicating one level of scope higher on the other. Equally, no distinction is made between delete (release previously allocated memory) and delete[] (release an array of such memory). This is, after all, a *Made Simple* book, and the precedence table above is quite complete enough for any practical purpose you are likely to have.

If you can remember the order of precedence and associativity for all operators in C++, fine. Otherwise *use parentheses*, even if they are not strictly necessary. It costs nothing to use the parentheses. It also saves errors and improves code readability.

Operators	Associativity
_ (scope resolution)	none
-> . (member selection)	left to right
++ (post increment)	none
— (post decrement)	[redacted]
[] (array subscript)	left to right
() (function call)	left to right
typeid() (find type of object)	none
const_cast (cast operator)	[redacted]
dynamic_cast (cast operator)	[redacted]
reinterpret_cast (cast operator)	none
static_cast (cast operator)	none
s zeof	none
++ (pre increment)	none
-- (pre decrement)	none
(type) (old-style cast operator)	right to left
new	[redacted]
delete	[redacted]
* (pointer dereference)	none
& (address-of)	none
+ (unary plus)	none
- (unary minus)	none
! (logical NOT)	none
* (pointer to class member)	right to left
->* (pointer to class member)	Right to left
* / %	left to right
+ -	left to right
<< >> (left- and right-bit-shift)	left to right
< > <= >=	left to right
== !=	left to right
& (bitwise AND)	left to right
^ (bitwise exclusive OR)	left to right
| (bitwise OR)	left to right
&&	left to right
||	left to right
?	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
throw (throw an exception)	none
, (comma operator)	left to right

Program example: validating a date

This section introduces the first program presented in this book that is actually useful. The program is called validate.cpp and does a job that every programmer in every language seems to have to do about 62 times in her life: checking that a given date is valid. The allowable date range is from the year 1901 to 2099. First, there is a header file, dates.h, that defines necessary preprocesser *symbolic constants* and holds function prototypes.

```
/**********************************************************************
 *
 *    'dates.h'
 *
 **********************************************************************/
#define MINYY   1901
#define MAXYY   2099
#define MINMM   1
#define MAXMM   12
#define MINDD   1
#define MAXDD   31
#define MINFEB  28
#define MAXFEB  29
#define TRUE    1
#define FALSE   0

// Function prototype declarations follow

void  get_data(int *, int *, int *);
int  validate(int, int, int);
```

Next, we have the program file validate.cpp, which contains the validation logic.

```
/**********************************************************************
 *
 *    'dates.cpp' — Program accepts as input a date of form
 *                  dd/mm/yyyy, validates the date, and returns
 *                  the result of the validation
 *
 **********************************************************************/
#include <iostream>
using namespace std;

#include "dates.h"

int main()
{
    int c, yy, mm, dd;
    get_data(&yy, &mm, &dd);

    // Check data for correctness. 1901-2099 date assumed
    if (validate(yy, mm, dd))
        cout << "Date entered is valid" << endl;
    [redacted]
        cout << "Invalid date entered" << endl;
}

void get_date(int *pyy, int *pmm, int *pdd)
{
    cout << "Enter day number ";
    cin >> *pdd
    cout << "Enter month number ";
    cin >> *pmm;
    cout << "Enter (four-digit) year number ";
    cin >> *pyy;
}

int validate(int yy, int mm, int dd)
{
    //     Validate the date entered according to the
    //     well-known rules

    if ((yy < MINYY) || (yy > MAXYY))
        return (FALSE);
    if ((mm < MINMM) || (mm > MAXMM))
        return (FALSE);
    if ((dd < MINDD) || (dd > MAXDD))
        return (FALSE);
    if ((mm==4) || (mm==6) || (mm==9) || (mm==11))
        if (dd > (MAXDD - 1))
          return (FALSE);

    //     If the month is February and the year is divisible
    //     evenly by 4, we have a leap year

    if (mm == 2)
    {
        if (dd > MAXFEB)
            return(FALSE);
        if (((yy % 4) != 0) || (yy == MINYY))
            if (dd > MINFEB)
              return(FALSE);
    }

    //     valid date
    return(TRUE);
}
```

Two functions are called from main: get_data and validate. get_data does what its name suggests: it prompts the user for input of three numbers, which are then treated as day, month and year respectively. get_data must set its parameters to the user-input data and the changed parameter values must be available in main after the call to get_data. To achieve this, get_data is called by reference: addresses of the arguments are used, not their values. Within get_data, the input stream object cin accepts data from the user into the memory at those addresses.

After the input date has been read by get_data, the three numbers are passed (by value) to validate for checking. There should be no need here to go in detail through the logic of validate; it's pretty clear and you should be able to understand it without further explanation. There is one small exception: why is the allowable year-range given as 1901-2099? Reason is that 1900 and 2100 aren't leap years, while 2000 is. Excluding 1900 and 2100 simplifies the logic of dates.cpp.

Depending on the TRUE/FALSE status returned by validate, a message confirming the validity, or not, of the date is output from main. Here's the screen output (user input in boldface) of two runs of the program:

```
Enter day number 29
Enter month number 02
Enter (four-digit) year number 2003
Invalid date entered

Enter day number 29
Enter month number 02
Enter (four-digit) year number 2004
Date entered is valid
```

validate.cpp pulls together in one program many of the important aspects of C++ syntax that you have seen so far in this book.

I've deliberately done this program without classes. Its purpose is to illustrate some of the other constructs and mechanisms of the C++ language – including functions, arguments, branching, comparisons and precedence – that you've seen up to this point. In Chapter 8, I use a class version of the same program to illustrate some of the simpler characteristics of C++ classes.

sizeof operator

You can use the sizeof operator when you need to know the size in bytes or characters in memory occupied by a data object.

In nearly all C++ environments, a char occupies the same amount of memory as an 8-bit byte, but the equivalence is machine-dependent and there are cases where this is not so. The sizes of other data objects — float, int and so on — are machine-dependent and no assumptions should be made about them when writing portable code.

In most cases, you don't want to know the actual number of bytes occupied by a particular data object. The object occupies a certain amount of space. You need to be able to access that value (without necessarily knowing what it is) so that you can use it later in your program.

The sizeof operator returns the size in bytes of its operand. If the operand is a type-specifier, it must be enclosed in parentheses; if it is a variable, the parentheses are optional. sizeof is used like this:

```
sizeof <variable name>
sizeof (<type specifier>);
```

Here are some examples of sizeof in use with typical data declarations and definitions:

```
char  c;
int i;
double  d;
float  f;
char  carr[10];
int  iarr[5];
char  *cptr;
int  *iptr = iarr;

sizeof(c)        // 1 by definition
sizeof(i)        // 4 if 32-bit system
sizeof(d)        // 8 if 32-bit system
sizeof(f)        // 4 if 32-bit system
sizeof(carr)     // 10: note the exception!
sizeof(iarr)     // 20 if 32-bit system
sizeof(cptr)     // 2 or 4
sizeof(iptr)     // 4 if 32-bit system

// type sizes
sizeof(int)      // 4 if 32-bit system
sizeof(char)     // 1 by definition
sizeof(float)    // 4 if 32-bit system
sizeof(double)   // 8 if 32-bit system
```

Suppose that we declare a simple structure:

```
struct sp_cell_s
{
    int       ival;
    double    dval;
    char      sval[20];
}
```

The value of sizeof(struct sp_cell_s) is at least 32 (the exact number depends on the compiler and the way it allocates memory) assuming a 4-byte integer, an 8-byte double, and adding the 20-byte array, plus the total of bytes, if any, needed for member alignment.

sizeof is special in that it is a *compile time operator* that yields a constant value. sizeof therefore can only yield information that is available to the compiler. It cannot know, for example, what the contents of a pointer will be at some point during program execution; it can only report the size of the pointer itself, not what it may in the future point to.

If the operand of sizeof is an array name, the extent of the memory occupied by the array is available at compile time thanks to specification of a constant-expression subscript limit. In this case, as an exception (see carr above), the array name is treated not as the address of the array but as representing the actual memory occupied by the array.

Exercises

1 Write a program that implements a preprocessor macro to display the minimum of two integer numbers.

2 The following variation of the function validate fails. Why?

```
int validate(int yy, int mm, int dd)
{
    //     Validate the date entered according to the
    //     well-known rules

    if ((yy < MINYY) || (yy > MAXYY))
        return (FALSE);
    if ((mm < MINMM) || (mm > MAXMM))
        return (FALSE);
    if ((dd < MINDD) || (dd > MAXDD))
        return (FALSE);

    if (mm==4 || mm==6 || mm==9 || mm==11
                && (dd > (MAXDD - 1)))
        return (FALSE);

    //     If the month is February and the year is divisible
    //     evenly by 4, we have a leap year

    if (mm == 2)
    {
        if (dd > MAXFEB)
            return(FALSE);
        if ((yy % 4) != 0)
            if (dd > MINFEB)
              return(FALSE);
    }

    //     valid date
    return(TRUE);
}
```

3 What happens when this statement

```
while (c = getchar() != 'q')
```

is executed?

4 Modify validate.cpp so that it checks for correctness all dates in the range 00 - Year zero to 9999. Be aware that 3 - 13 September 1752 (inclusive of both days) were lost in the switch from the Julian to the Gregorian calendar. Also, years divisible with zero remainder by 4 AND by 400 are leap years. Years divisible evenly by 100 but NOT by 400 are not leap years. 1900 was therefore NOT a leap year.

6 Program flow control

Program structure

As noted in Chapter 3, the body of a function following the function header is a compound statement or statement block.

C++ is an object-oriented (OO) language, derived from the earlier C language. C++'s fundamental structure reflects that object-orientation: the central constructs in any program are classes (or templates); these contain function members whose code specifies the logic of operations to be performed both on other class members and on outside entities. At the start of Chapter 1 there is a brief summary of the OO facilities of C++ and of the OO development approach.

Considered in a more limited way, C++ is also *block-structured*. Its OO emphasis accepted, C++ also reflects the characteristics of C. This chapter concentrates on those aspects of block structure and the flow of execution control between the different code blocks.

In addition to promoting OO design, C++ encourages programs to be written according to the rather loosely defined principles of *structured programming*. In the original definitions of languages such as COBOL and FORTRAN, there was no inherent structured approach. Change in the order of execution of statements in the program (control flow) was accomplished using an unconditional branch statement (GOTO) or a subroutine call (such as CALL, PERFORM).

Unstructured control flow makes for unreadable code. This is inefficient, prone to errors on the part of the programmer and, as an unstructured program grows, increasingly difficult to maintain.

The following are some simple principles of the structured programming approach:

- Programs are designed in a top-down manner; the major functions required for the solution are called from the highest-level function. Each of the called functions further refines the solution and calls further functions as necessary.
- Each function is short and carries out one logical task.
- Every function is as independent as possible of all other functions. Information is exchanged between functions via arguments and return values. Use of global variables and shared code is minimised.
- Unconditional branching is avoided.

C++ provides the facilities necessary to meet these objectives. All statements are either simple statements – expressions terminated by semicolons – or compound statements, which are statement groups enclosed by curly braces {}. A compound statement is syntactically equivalent to a simple statement.

Every C++ program must have a main function in which, ideally, instances of important classes are created. Both from the main function and from the member functions of these classes, further functions, representing lower levels of the solution, are typically called.

It is sometimes held that no function should be longer than 50 lines of code. If it is, it should be broken down into a calling and one or more called functions.

Using classes, functions, return values and arguments, C++ allows you to exchange data between functions to an extent which minimises use of global variables. C++ does provide a goto statement for unconditional branching, but its use and power are severely restricted.

C++ provides a range of statements for control of program execution flow. These are all based on switching control between the program's constituent compound statements and, collectively, they allow you to write concise, logically structured programs.

Conditional branching

if

The general form of the if statement is this:

```
if (<expression>)
    <statement1>
[else
    <statement2>]
```

The square brackets indicate that the else clause is optional.

The expression may be any legal expression, including an assignment, function call or arithmetic expression. The inherent boolean value of the expression determines change, if any, made to the program's flow of control by the if statement.

The statements subject to the if and the else may be any legal single or compound statement. If a single statement is subject to an if or else, use of the compound-statement delimiters {} is optional; for two or more statements they are necessary. For example, when getchar reads the letter 'q' from the keyboard, we can stop program execution:

```
if ((c = getchar()) == 'q')
{
    /* Finish program execution */

    cout << "Program terminating\n" << endl;
    return(0);
}
```

You can nest if statements and the optional else clauses to any depth:

```
if (mm == 2)
    if ((yy %4) != 0)
        if (dd > MINFEB)
            return(FALSE);
```

What does this triple-nested if mean? If the month is February AND if the year is not a leap year AND if the day is greater than 28, there is an error.

Each if statement, including its subject compound statement(s), is syntactically a single statement. This is why the last example, although it contains three nested if statements, is a single statement; no compound statement delimiters, {}, are necessary.

You can use the delimiters if you like:

```
if (mm == 2)
{
    if ((yy %4) != 0)
    {
        if (dd > MINFEB)
```

```
            return(FALSE);
        }
    }
```

This makes no difference at all to the logic but gives an improvement in code readability. Use of compound statement braces becomes important when the else option is used.

```
    if (mm == 2)
        if ((yy %4) != 0)
            if (dd > MINFEB)
                return(FALSE);
    else
        return(TRUE);  //        valid date
```

Each else corresponds to the last if statement for which there is no other else, unless forced to correspond otherwise by means of {} braces. In this example, the else refers to the third if, although it is presumably intended to correspond with the first. To get the result you probably want, write this:

```
    if (mm == 2)
    {
        if ((yy %4) != 0)
            if (dd > MINFEB)
                return(FALSE);
    }
    else
        return(TRUE);  //        valid date
```

Lastly, you can nest to any depth the whole if...else construct itself:

```
    if (dd == 1)
        cout << "Monday" << endl;
    else
    if (dd == 2)
        cout << "Tuesday" << endl;
    else
    if (dd == 3)
        cout << "Wednesday" << endl;
    else
    if (dd == 4)
        cout << "Thursday" << endl;
    else
    if (dd == 5)
        cout << "Friday" << endl;
    else
    if (dd == 6)
        cout << "Saturday" << endl;
    else
    if (dd == 0)
```

```
        cout << "Sunday" << endl;
    else
        cout << "Error" << endl;
```

This is a multi-way decision. The construct is more efficient than it would be if all the if statements were used without else clauses. As it stands, as soon as an individual test is successful, execution of the whole sequence stops. If dd is not in the 0-6 range, the last else does processing for the "none of the above" case and flags an error. There is a special facility in C++ for more efficiently handling cases like this. I explain the switch statement in *Multi-case selection*, page 139.

Loops

Chapter 1 describes the basic rules governing the three loop constructs available in C++. Here, I present a single example, implemented in turn with each of the loop types.

The first code sample uses the while loop:

```
cout << "Press RETURN to start, 'q'-RETURN to quit "
while ((c = getchar()) != 'q')
{
    // call all the top-level functions of
    // the program
    cout << "Press RETURN to start, 'q'-RETURN to quit "
}
```

Next, here's the equivalent functionality in a for loop

```
for ( cout << "Press RETURN to start, 'q'-RETURN to quit "
    c = getchar();

    c != 'q'

    cout << "Press RETURN to start, 'q'-RETURN to quit "
    c = getchar();
    )
{
    // call all the top-level functions of the program
}
```

Note that, in the for loop, there are three governing expressions.

```
for (<expr1>;<expr2>;<expr3>)
```

where the three expressions are (and must be) separated by two semicolons. It's possible, as in the example above, to have a single expression made up of a number of comma-separated subexpressions. You can see this in the two cases where a cout is followed by a getchar within a single expression. If, nonetheless, you suspect this is a bit clumsy and not the best use for a for loop, you're right

Lastly, here is similar code written with a do-while construct

```
do
{
    cout << "Press RETURN to start, 'q'-RETURN to quit "
    c = getchar();

} while ((c = getchar()) != 'q')
```

The requirements of this logic naturally suit use of the while loop type. The for loop works also, but is cumbersome because all the loop-controlling code and some unrelated prompt code must be grouped at the increment step at the top of the loop

It is best to restrict the increment step to loop-control code and put other statements into the body of the loop.

The do-while variant does not produce quite the same result as the other two. There is an initial prompt and an initial character is read, whether or not it is 'q'. Only on subsequent entry of the keystroke 'q' does the loop terminate.

In general, if an operation can be implemented with one loop construct, it can also be done with the other two. Usually, however, one of the three types will be most suitable. The for loop suits cases where the number of iterations is known in advance, as in the cases of traversing an array with fixed subscript limits and reading a data stream until end-of-file. The while loop is best for doing something until an external condition (e.g. keystroke 'q' for quit) arises. The do-while construct is appropriate where a loop must be executed at least once, for example when a menu must be displayed.

It is always possible to use while and for interchangeably, but while is usually suitable in those cases where for is not. The general form of the for statement is:

```
for (<expr1>;<expr2>;<expr3>)
    <statement>
```

where <expr1> is the list of initialising expressions, <expr2> is an expression list controlling loop termination, and <expr3> is the so-called increment step. This is equivalent to:

```
<expr1>
while (<expr2>)
{
    <statement>
    <expr3>
}
```

The exception is when the continue statement is used to change the flow of control of execution in the loop. continue is described in the next section.

do-while is useful where it is required always to do one iteration before the controlling condition is tested. In the case above, this is not so and, although the code works, use of the do-while is inappropriate. Note that the do-while loop must be terminated with a semicolon.

The two while loop types must update the variable whose value is controlling exit from the loop. The variable is updated either in the loop's compound statement or in the controlling expression. Care needs to taken because the controlling expression is evaluated in a different place in the two loop types.

Finally, all three loop statements are syntactically single statements. They may therefore be nested to an arbitrary level without using compound statement delimiters.

```
for (int i=0;i<100 && arr1[i];i++)
    for (int j=0;j<100 && arr2[j];j++)
        if (arr1[i] == arr2[j])
            return(i);
```

The whole construct is a single statement that compares every element of arr1 with every element of arr2, stopping if a match is found. This is potentially up to 10,000 loops, so you should (from a standpoint of efficiency) take care when nesting loops and consider whether there isn't a less "brute-force" solution.

Note also that you can declare the variables i and j as part of the initialising (first) expression in the for loop. This is common practice in C++.

Unconditional branch statements

There are four unconditional branch statements available in C++. In ascending order of power, they are:

- break
- continue
- return
- goto

You will sometimes want to exit a loop before the controlling condition causes the loop to end normally. The break and continue statements, in different ways, allow this to happen.

break

You can use the break statement to exit a loop early. To illustrate, let's define an character array of ten elements and an array subscript (in the for loop initialising expression):

```
char arr[10];
```

Assume the array has been initialised with a string. First, we traverse the array, display each character and exit the loop early on encountering a '\0' character:

```
for (int sub = 0; sub < 10; sub++)
{
    if (arr[sub] == '\0')
        break;

    cout << arr[sub] << endl;
}
```

When break is encountered, it causes unconditional exit from the loop. Control is transferred to the first statement following the loop's compound statement. break only causes exit from one level of loop: if the loops are nested, control is returned from the loop containing the break to the outer loop. You can use break within any of the loop types as well as with the switch statement, seen in the next section.

continue

You can use continue to skip iterations within a normal loop sequence, but not to exit the loop altogether. We can illustrate continue using the same array and subscript counter defined above. Again, we want to traverse the array. If a newline character '\n' is encountered, it is ignored and the characters, stripped of newlines, are displayed.

```
sub = -1
while (sub < 10)
```

```
{
    sub++;
    if (arr[sub] == '\n')
        continue;
    cout << arr[sub] << endl;
}
```

When continue is encountered, it causes control to be passed to the loop's controlling expression or, in the case of a for loop, the increment step. If the loop does not terminate naturally, the next iteration is performed. continue can only be used within loop statements.

You should be able to see from the example one of the dangers of using continue. Because continue causes part of an iteration to be skipped, problems arise if the loop-control variable is updated during the skipped part. In the case above, if sub were incremented after the cout – as in ordinary code it probably would be – it would fail to be incremented for the first '\n' encountered. An infinite loop is what you get.

goto

The unconditional branch instruction goto may occasionally be useful but is never necessary. Anything that you can accomplish with goto you can also do with combinations of the flow-control statements shown earlier in this chapter.

Control is transferred unconditionally from the goto statement to the point in the code where a named label followed by a colon is encountered. Use of goto is not recommended; it tends to lead to undisciplined and unreadable code. However, there are a few cases where it serves a purpose.

The break statement causes exit from one loop to the first statement in the code surrounding the loop. return causes control to be returned from a function to the statement after the function call in the calling function. Where loops are nested two or more levels deep, there is no ready way to transfer control from the innermost loop to a point outside all the nested loops but without leaving the function.

Here is an example of a reasonable use of goto. We define two character arrays:

```
char arr1[100];
char arr2[100];
```

Assume the two arrays have been initialised with C-strings. We want to find a character in arr2 that also exists in arr1 and then to exit:

```
for (int i=0;i<100 && arr1[i];i++)
{
    for (int j=0;j<100 && arr2[j];j++)
    {
        if (arr1[i] == arr2[j])
```

```
            goto match;
    }
}
cout << "No match found" << endl;
goto end;
match: cout << "Match found " << arr1[i] << endl;
end:        ;        // null statement
```

The goto label is only visible within the function containing the goto statement. You can therefore only use goto within one function. This is the definition of 'function scope', the only one of the five kinds of C++ scope not explained in *Storage class and scope* in Chapter 3.

goto should not be used to transfer control to a statement within a loop. If the loop-control variables have already been initialised, use of goto to a point in the middle of the loop may bypass that initialisation and the loop will go out of control, probably with unpleasant results.

Multi-case selection

As I've already pointed out C++ provides a statement to handle the special case of a multi-way decision. Here is the switch implementation of the if else multi-way decision given earlier in this chapter

```
switch(dd)
{
    case 1.   cout << "Monday" << end
              break
    case 2:   cout << "Tuesday" << endl.
              break,
    case 3:   cout << "Wednesday" << endl:
              break,
    case 4.   cout << "Thursday" << endl
              break.
    case 5:   cout << "Friday" << endl:
              break.
    case 6:   cout << "Saturday" << endl:
              break
    case 0:   cout << "Sunday" << endl.
              break
    default   cout << "Error" << endl
              break.
}
```

Execution control is switched depending on the value of the variable dd The variable must be of one of the integer types or of type char Each of the expected values of dd is enumerated. If dd is one of those values, the code adjacent to the case label is executed The case values must be constants All case expressions in a given switch statement must be unique.

switch in C++ provides entry points to a block of statements When control is transferred to a given entry point execution starts at the first statement after that entry point Unless directed otherwise execution will simply fall through all code following even though that code is apparently associated with other case labels

For this reason you need to insert a break at the end of the statements subject to a case label unless you want all the code within the switch block starting from a given case label, to be executed.

In the example above if all the break statements were left out and dd had the value 3, control would fall through the switch statement to the end and messages for all the days from Wednesday through to Sunday would be displayed. Omit break statements from switch at your peril!

If you use break in a switch statement it causes control to be transferred completely out of the switch statement continue does not apply to switch it only has any effect if the switch statement is embedded in a loop

The default case prefixes code which is executed if none of the previous case conditions is true You should end the statements subject to default with a break

Inclusion of the default case is optional. There must be only one default in a switch statement (or none). default may occur anywhere in a switch statement, but is usually placed at the end.

If the statements labelled by a case immediately preceding the default label are executed, control will fall through to the default label unless a break statement is encountered. You can nest switch statements any depth. A case or default label is part of the smallest switch that encloses it.

There follows a somewhat contrived program, jumpstmt.cpp, notable mainly for the fact that it succeeds in using all the unconditional branch statements together.

```
/*************************************************************
 *
 *    'jumpstmt.cpp' — Program repeatedly accepts as input a
 *                character and tests it for being a number
 *                in the range 1-7 for a day of the week.
 *
 *************************************************************/
#include <iostream>
using namespace std;

#include <cstdio>

int main()
{
    int c;

    cout << "Enter a single-digit number: ";
    while ((c = getchar()) != EOF)
    {
        if ((c == '\n') || (c == '\r'))
            continue;
        if ((c < '0') || (c > '9'))
        {
            cout <<
            "You must enter a single-digit number" << endl;
            cout << "Enter a number: ";
            continue;
        }
        switch(c)
        {
            case '8':
            case '9': cout << "Number not in range 1-7 " << endl;
                 break;
            default:  goto finish;
            case '1': cout << "Monday " << endl;
                  break;
            case '2': cout << "Tuesday " << endl;
                  break;
            case '3': cout << "Wednesday " << endl;
                  break;
            case '4': cout << "Thursday " << endl;
                  break;
            case '5': cout << "Friday " << endl;
                  break;
            case '6': cout << "Saturday " << endl;
                  break;
            case '7': cout << "Sunday " << endl;
                  break;
        }
        cout << "Enter a number: ";
    }
    goto returnnow;
finish:
    cout << "Zero invalid, program terminating. " << endl;
returnnow:
    return 0;
}
```

At the start of the main function, the user is prompted to enter a number intended to represent a day of the week. When the user presses RETURN, newline ('\n') and carriage return ('\r') characters are also generated at the keyboard, so the program discards those by using **continue** to go back to the top of the loop and get another character. If the character is not in the range zero to 9, the code similarly transfers control to the top of the loop.

Finally, there is a switch statement. Cases 8 and 9 are discarded as invalid. The **default** case (which need not be at the end of the switch) uses a **goto** to transfer control to the label finish and a message proclaiming input of zero to be invalid. After the switch construct, another **goto** is used to jump over the finish label; this gives an insight to the kind of spaghetti code you can generate using **gotos** if you're not careful. When you run the program and enter data as prompted, you get a screen display something like this (user input in boldface):

```
Enter a single-digit number: 5
Friday
Enter a number: 4
Thursday
Enter a number: 9
Number not in range 1-7
Enter a number: 0
Zero invalid, program terminating.
```

Exercises

1. Using each of the three loop forms, write infinite loops.
2. Write a program that presents a simple menu of five numbered items and then waits for the user to enter a number selecting one of them. Display acknowledgement of the selection or report an error if the selection is not in the range 1–5.
3. Write a program that, for the 20th and 21st centuries, finds the *day of the date* for a date input as the three separate numbers *dd*, *mm* and *yy*. (This is not a trivial problem: allow yourself several hours.)

7 Memory management

Linked structures

One of the major uses of pointers is in allowing construction of advanced data structures such as linked lists. Linked lists are built from structure instances connected by pointers. A structure must not contain a nested instance of itself, but it may contain a pointer to one of its own kind. This allows us to build chains of structure instances that are, in fact, lists. (Note that the C++ Standard Library (see Chapter 12) provides templates and classes that uniformly implement not only linked lists but also other data structures including queues and sets. These effectively hide from the programmer the pointer-level manipulation and memory-allocation operations that you're going to see in this chapter. Don't think that availability of the STL removes the need for you to know the fundamentals of pointers, lists and memory management.)

The following paragraphs gently introduce the mechanisms used in manipulation of linked lists. I use a pair of structure instances, defined in the conventional (non-dynamic) manner. The structure instances contain pointers of their own type, allowing one to be linked to the other by containing the address of the other. The address link is used to traverse the simple two-element list. Coverage of dynamic memory allocation is deferred until the section *Dynamic storage allocation* below.

Here is an example of a structure declaration containing a pointer to another structure of the same type:

```
struct node
{
    int x;
    double   y;
    node *next;
};
```

next is a pointer to a data object of type **node**. Let's define two instances of this structure:

```
node first, second;
```

We assign values to the structure members like this:

```
first.x    = 5;
first.y    = 34.78;
second.x = 6;
second.y = 45.89;
```

and now link the structures by assigning the address of the second to the pointer member of the first:

```
first.next = &second;
```

After the address assignment, next is the address of the second structure and the structures' members can be accessed like this:

```
first.x          // 5
first.y          // 34.78
second.x         // 6
second.y         // 45.89
first.next->x    // 6
first.next->y    // 45.89
second.next      // indeterminate value
                 // should be set to NULL
```

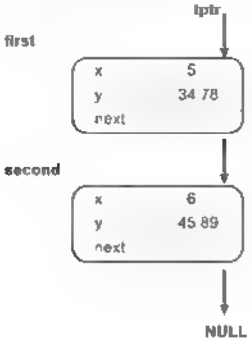
Suppose that we define and initialise a pointer to a structure of type node

```
node *tptr = &first;
```

Now the members of the two structures can be accessed using the pointer notation

```
tptr->x          // 5
tptr->y          // 34.78
tptr->next->x    // 6
tptr->next->y    // 45.89
```

The list's organisation can be depicted graphically:



Programmer-defined data types

typedef specifier

As you've seen, you can use structures to define a data type that is a combination of C++'s basic data types. So far, you have seen four storage class specifiers:

```
auto      extern
static    register
```

There is a fifth, typedef, which allows you to define original data types of arbitrary complexity. typedef fits uneasily as a member of C++'s storage class specifiers. It is really a method by which you can define new type specifiers in terms of existing types.

typedef, as well as being a way of defining new types, is a useful shorthand. Type definitions made using typedef are usually grouped in header files and used throughout the program as a type specifier like any other.

Assume that all prices are stored as double floating-point numbers. We can then make the following type definition:

```
typedef double price;
```

and define instances of the new type:

```
price cost_price;
price sell_price;
```

More-complex type definitions

The trivial example above serves no real purpose other than, perhaps, to improve program readability. We are interested in more complex definitions. Let's look at a simple (no function members) structure declaration stock_type:

```
struct stock_type
{
   char      item_name[30];
   char      part_number[10];
   double    cost_price;
   double    sell_price;
   int       stock_on_hand;
   int       reorder_level;
};
```

The whole structure can be given a type-name of its own using typedef:

```
typedef struct stock_type
{
   char      item_name[30];
   char      part_number[10];
   double    cost_price;
```

```
        double    sell_price;
        int       stock_on_hand;
        int       reorder_level;
    }ITEM
```

In this case ITEM is not a definition of an instance of the structure type stock_type but instead becomes a synonym for the data type stock_type. Now we can define instances of the structure using the shorthand:

```
    ITEM item1, item2;
```

Using the self-referencing structure declaration given in the first section of this chapter:

```
    typedef struct node
    {
        int       x;
        double    y;
        node      *next
    }NODE
```

NODE is now a data type specifying the structure type node. To define two of these structures and a structure pointer, we make the definitions and initialisation:

```
    NODE first, second, *tptr = &first;
```

tptr may now be used as at the start of this chapter to reference the members of the two structures.

Specification of new types based on structures is very common. It is typical to provide a type definition for both a structure instance and a pointer of the structure's type:

```
    typedef struct node
    {
        int       x;
        double    y;
        node      *next
    }NODE, *PNODE;
```

Once these types have been defined and incorporated in a header file, you can use pointers of the structure's type without having to be concerned with the asterisk pointer notation:

```
    NODE inst;
    PNODE nptr = &inst
```

The array type

A complex use of typedef that is often not understood is this:

```
        typedef char array_type[256];
```

Here, array_type is not the name of a character array of 256 elements; it is rather a type representing character-arrays-of-256-elements. It can be used to define actual arrays of 256 elements.

```
        array_type a1_256 = "Three thousand ducats, a good round sum";
```

such that the following statement

```
        cout << a1_256[7] << endl;
```

yields the character 'h'.

Preprocessor and typedef compared

Another use of typedef that some programmers favour is this:

```
    typedef char *charptr;
```

Now charptr is a synonym for char * and you can use it to define character pointers.

```
    charptr cptr1, cptr2;
```

It is important to understand the meaning of the different shorthand mechanisms provided by C++. For example, use of typedef and the preprocessor may provide superficially similar results. In the following case, both char1 and char2 are character pointers.

```
    #define PCHAR char *
    typedef char * charptr;

    PCHAR      char1;
    charptr    char2;
```

but the next definitions expose the difference between pattern substitution as implemented by the preprocessor and the true type synonym provided by typedef:

```
        // substitutes to: char *char1, char2;
    PCHAR char1, char2;

    // correctly defined as char *char1, *char2;
        charptr char1, char2;
```

Portability

typedef can be useful in producing portable programs:

```
        typedef long INT;
        INT portable_int;
```

portable_int is a long integer on both 32- and 64-bit systems. Notation like this is widely used to ensure that code written for a 64-bit operating system will be portable without change to older environments such as today's Windows systems and many of today's UNIX variants.

Dynamic storage allocation

Operators and functions

Up to now, the only way we have seen of allocating memory space to a variable is by definition of that variable and allocation of space by the compiler. All variables defined up to this point have been of fixed length and the allocation of space has been outside the control of the programmer.

To store repeated instances of aggregate data (or 'records') we could use an array of structures. However, arrays themselves are of fixed length, determined at compile time. If the records were being generated from data entered at a device such as a terminal, then no matter how large the array defined to store the data, it might not be large enough.

What is needed is a way of allocating memory, under the control of the programmer, at program run time. This is accomplished using the new and delete operators of C++ or, alternatively, the older equivalent functions of the Standard C Library, the prototypes of which are available in the standard header file cstdlib.

The new and delete operators have superseded the C library functions, which are now regarded as at least obsolescent. However, there's a lot of code out there that uses the C functions, so I'll mention them here before moving on to concentrate on new and delete. The four functions are these (declared in header file cstdlib):

malloc returns a pointer to a specified amount of memory, which is allocated from the program heap by the C++ dynamic allocation system.

calloc does the same as malloc, but returns a pointer to an array of allocated memory and initialises that memory with zeros.

realloc changes the extent of memory allocated by malloc or calloc and associated with a pointer to a specified size, while preserving the original contents.

free frees allocated memory and makes it available to the system heap.

For the prototypes and details of operation of these functions, see Chapter 14.

Dynamic storage allocation must be used in any situation in which you don't know in advance how much data will be entered to a program. Such an eventuality usually takes either of two forms:

- data records are entered by an operator or another program; the receiving program uses dynamic allocation of a structure for each unit of record data entered to create a list or file of effectively unlimited length.
- a text string entered by an operator is stored in a large input buffer of the maximum possible line length (say 512 characters). To record each line in a page of text as 512 characters is wasteful of memory, so dynamic allocation is used to set each line's pointer pointing only to enough memory to record the actual text and a null-terminating character.

- Later in this section I explain how to use dynamic allocation to implement the linked structures shown earlier. For a full-blown linked list done with dynamic allocation, see my own *C++ Users Handbook* or any of many other titles that deal with data structures. Before we can look at even the limited list application, however, we have to explore the operation of **new** and **delete**.

Using new and delete

new and **delete** are the C++ replacement operators for the C library functions **malloc** and **free**. You can still use **malloc** and **free**, but **new** and **delete** are easier and safer to use.

The **new** operator is almost always used in one of the following general forms:

```
<ptr> = new <type>(<initial value>)

<ptr> = new <type>[<size>];
```

The angle brackets here indicate that the value within is replaced by an actual literal value. The first example allocates space for one instance of a given type and sets it to an initial value. The second allocates space for an array of objects of a given type. **new** returns to the pointer on the left-hand side of the assignment a pointer to the memory allocated. The pointer is of the type specified on the right-hand side of the assignment. If for any reason the memory cannot be allocated, **new** returns a NULL pointer, which may be used in application code to check for a memory allocation error.

The general forms of the **delete** operator are these:

```
delete <ptr>

delete [] <ptr>;
```

These deallocate the space pointed to by the pointer **<ptr>** which was previously allocated by **new**. Use of a pair of square brackets in the second case indicates that the memory to be deallocated was allocated in the first place as an array using **new[]**.

Here is a program that summarises these uses of **new** and **delete**:

```
#include <iostream>
using namespace std;

int main()
{
     int *iptr1, *iptr2, *iptr3;

     // allocate memory for two individual integers
     // and an array of 20 integers
     iptr1 = new int (5);
```

```
        iptr2 = new (int) (6);
        if ((iptr3 = new int[20]) == NULL)
            cout << "Couldn't allocate array" << endl;
        else
            *iptr3 = 25;

        //  Display first two integer values
        cout << "Integer 1: " << *iptr1 << endl;
        cout << "Integer 2: " << *iptr2 << endl;
        cout << "Integer 3: " << *iptr3 << endl;

        // deallocate memory at all 3 pointers
        delete (iptr1);
        delete iptr2;
        delete [] iptr3;
    }
```

The displayed results are

```
5
6
25
```

These are the values of the dynamically allocated integer variables pointed to by iptr1 iptr2 and iptr3

Allocating a list element

Let's look again at the structure declaration:

```
typedef struct node
{
    int         x
    double      y
    node        *next
}NODE
```

Recall that NODE is not a structure instance but a typedef giving a new, programmer defined type specifier representing a structure of type node

Let's now allocate enough space for such a structure

```
NODE *ptr1
ptr1 =  new NODE,
```

After this, ptr1 points to an instance in memory of structure type NODE If there isn t enough memory available for the allocation or if there is some other error new returns NULL to ptr1. This leads us to the complete construct for allocation of memory:

```
if ((ptr1 = new NODE) == NULL)
{
    cout << "Memory allocation error" << endl;
    exit(0);    // Exit program
}
// Memory successfully allocated
```

You'll find the prototype for the exit library function in the header file cstdlib. It causes graceful program termination and returns a status code to the local operating system environment. Zero indicates a successful termination.

Freeing allocated memory

The most common and serious error made in C++ programs is that of using a pointer before it has been set pointing to an allocated memory object.

A good candidate for second place in the league of C++ programming errors is failure to release allocated memory when it is no longer required. Failure to free allocated memory is actually more insidious than use of an uninitialised pointer. Using such a pointer causes the program to crash immediately; the error is therefore not difficult to find, especially with a good program debugging tool.

If you don't subsequently deallocate dynamically-allocated memory with the operator delete, the memory is not returned to the available memory pool, even when the program stops execution. This leads to a situation where the system gradually runs out of available memory, often resulting in a program crash far from the scene of the point where memory should have been freed. It can be extremely difficult to track down the source of such a *memory leak*.

For every dynamic memory allocation in a program, there should be a corresponding use of delete to make available the memory associated with the pointer. If the pointer ptr1 is associated with memory dynamically allocated by new, that memory is deallocated by the simple function call.

```
delete ptr1;
```

Memory deallocation is usually done when that memory is no longer needed, for example when a list element or line of text is deleted. There are two ways in C++ of ensuring that every allocation with new has a corresponding delete: use of class destructor functions, seen in Chapter 9, and the auto_ptr template, new to ISO C++.

Freeing memory with auto_ptr

ISO C++ provides a variant on the use of new that ensures that the allocated memory is released when the pointer to that memory goes out of scope. The use of auto_ptr is illustrated by the program autoptr.cpp.

```
#include <iostream>
using namespace std;
#include <memory>

typedef struct node
{
    int         x;
    double      y;
    node        *next;
}NODE;

int main()
{
    // allocate node pointer, memory freed when
    // pointer goes out of scope
    auto_ptr<NODE> ptr1(new NODE);

    ptr1->x = 5;                // OK
    ptr1->y = 3.14;             // OK

    auto_ptr<NODE> ptr2(new NODE[20]);      // ERROR
    ptr2+=10;                               // ERROR
    // ptr1 memory deallocated here
}
```

The operation of this program is the same as if the simple new were used, with the important addition that the allocated memory is released on exit from the main function. Note that auto_ptr can't be used as with ptr2 for dynamically-allocated arrays. Also, you can't do arithmetic with ptr2, as you can with ordinary pointers.

Dynamic allocation of list nodes

Now we know enough to do a second version of the linked structures shown at the start of this chapter. There, the two instances of the structure type node (later typedefed to NODE) are allocated with a conventional definition:

```
node first, second;
```

With dynamic allocation, the space is reserved using this code:

```
NODE *lptr1, *lptr2;

if ((lptr1 = new NODE) == NULL)
{
    cout << "Memory error allocating first node" << endl;
    exit(0);    // Exit program
}
```

```
if ((tptr2 = new NODE) == NULL)
{
    cout << "Memory error allocating second node" << endl;
    exit(0);    // Exit program
}
```

Now we can make the assignments of values to the structure instances' members:

```
tptr1->next = tptr2; // Link the two structures

    // Now make assignments to the structure members
tptr1->x   = 5
tptr1->y   = 34.78
tptr2->x   = 6
tptr2->y   = 45.89
tptr2->next  = NULL
```

In a real implementation of a list program, the dynamic allocation code and the assignments would be in a loop controlled by the user's input. In addition, you would have to manage the links between the list's members as well as strategies for insertion in and deletion from the list.

Given the simple use of new in this case (as opposed to auto_ptr), you have to remember to deallocate the memory explicitly:

```
delete tptr1
delete tptr2
```

Address arithmetic

You can do address arithmetic on pointers, usually with pointers to arrays. You've already seen a few typical cases of address arithmetic, where the displacement of a character pointer from its start point needs to be calculated in order to return a relative position in a C-string. Also, you should have become accustomed to the practice of repeatedly incrementing pointer values by one when traversing an array.

Let ptr be a pointer to an array of elements of some type. ptr++ increments the pointer to the next element in the array. *ptr is the contents of the element currently pointed to. ptr += n increments the pointer by the value of n array elements.

Each element of a character array is, by definition, one char or byte long. It's reasonable to expect a pointer to such an array to be incremented by one to point to the next element. In fact, for all arrays of any type of element, the 'increment by one' rule holds. The size of each element is automatically taken into account and it is a mistake, when incrementing the array pointer, to try to calculate the size of the array elements and increment by that amount.

To summarise: incrementing a pointer by one makes it point to the next element for all arrays, regardless of the type of the elements.

Address arithmetic example

Look again at the structure declaration struct stock_type

```
struct stock_type
{
    char      item_name[30];
    char      part_number[10];
    double    cost_price;
    double    sell_price;
    int       stock_on_hand;
    int       reorder_level;
}
```

Now we define an array of these structures and initialise a pointer to the array

```
stock_type stockarr[100];
stock_type *stockptr = stockarr;
```

Even though each array element occupies at least 50 bytes on any system, the pointer need only be repeatedly incremented by one to traverse the array

```
for (int count = 0; count < 100; count++,stockptr++)
{
    // Set the array elements zero
    // or empty

    stockptr->item_name[0]    = '\0';
    stockptr->part_number[0] = '\0';
    stockptr->cost_price      = 0.0;
```

```
        stockptr->sell_price     = 0.0;
        stockptr->stock_on_hand  = 0;
        stockptr->reorder_level  = 0;
    }
```

When two pointers to an array are subtracted, the result is not the number of bytes that separate the array elements but the number of array elements.

You absolutely shouldn't do arithmetic of this kind on pointers of different types: the results will be unpredictable and probably catastrophic. Two pointers of the same type may be subtracted but *not added, divided or multiplied*. Addition to a pointer is only legal where the pointer is incremented by an integral (small whole-number) value.

Precedence and associativity

You need to be careful with the syntax of pointer increment and decrement operations. The ++, -- and * (dereferencing) operators are all of the same precedence and associate right to left. As a result, some unexpected things can happen when these operators are mixed in the same expression. For example, *ptr++ is a very common expression. It means that the object at ptr is fetched (first) and then (second) the *pointer value* is incremented by one. If we want to add one to the *object at the pointer*, we need (*ptr)++.

Liberal use of parentheses in the case of mixed-operator expressions like that above and care about not mixing pointer types will save a lot of trouble. The program ptrinc.cpp illustrates the point.

```
/*********************************************************************
*
*     'ptrinc.cpp' — Program to illustrate compound pointer arithmetic
*
*********************************************************************/
#include <iostream>
using namespace std;

int main()
{
    char stg[] = "nmlkjihgfedcba";
    char *ptr = stg;

    cout << "Initial string is " << ptr << endl;
    cout << "Display and post-increment the pointer" << endl;
    cout << "*ptr++ " << *ptr++ << endl;
    cout << "*ptr " << *ptr << endl;
    cout << "Re-initialise pointer" << endl;
    ptr = stg;
    cout <<
    "Display and post-increment the OBJECT AT the pointer" << endl;
    cout << "(*ptr)++ " << (*ptr)++ << endl;
    cout << "*ptr " << *ptr << endl;

    /* Results to be expected

      *ptr++   n
      *ptr     m

      (*ptr)++ n
      *ptr     o   */
}
```

The essence of this program is the fact that *ptr++ retrieves data and then increments the pointer ptr while (*ptr)++ adds 1 to the data ('n'), giving 'o'. When you run the program, you get this screen display:

```
Initial string is nmlkjihgfedcba


Display and post-increment the pointer
*ptr++ n
*ptr m

Re-initialise pointer
Display and post-increment the OBJECT AT the pointer
(*ptr)++ n
*ptr o
```

*ptr++ is probably one of the most common forms of expression used in all C++ programming so, again, don't get the idea that this pointer arithmetic stuff is for nerds. It's part of everyday C++ programming, can become very complex, and you need to be adept at it.

Arrays of pointers

It's possible in C++ to define a pointer to a pointer (or a pointer to a pointer to a pointer if you feel the inclinations. Pointers to pointers are sometimes called *multiply-indirected* pointers. An important application of multiply indirected pointers is in accessing and traversing multidimensional arrays. Implementing an N-dimensional array in C++ using pointers requires definition of a pointer array of N-1 dimensions. In the case of a two-dimensional character array, which can store a page of text, we must define a one-dimensional array of pointers of type char *.

To do this, we define an array of character pointers:

```
char *cptr[10];
```

Each of the pointers in the array must be initialised to the address of an array of characters before being used.

```
char *cptr[10] = {"Signor Antonio, many a time and oft\n",
                  "on the Rialto, you have rated me\n",
                  "for my moneys and my usances.\n",
                  "Still have I borne it with a patient shrug,"
                  "for sufferance is the badge of all our tribe.", ""};
```

In this case, pointers zero to 4 of the ten-element pointer array are initialised to the addresses of the five literal C-strings shown within curly braces.

cptr[2] points to the string "on the Rialto, you have rated me\n". Instead of using subscripts to access array elements, we can use a pointer to the array of pointers:

```
char **cpp = cptr;
```

After the pointer initialisation,

*cpp points to the C-string "Signor Antonio, many a time and oft\n"

**cpp is the first character in that string, 'S'

(*cpp)++ increments the pointer to the first C-string.

**cpp is now the second character, 'i'

Program example: array2d.cpp.

Here is an example program that exercises many of the possible operations using a pointer to pointers on a two-dimensional character array. The array of pointers is defined and initialised to its six component C-strings.

The program performs two principal operations. The first displays each of the text lines in turn with subscripts, the second does the same with pointers. The iteration is terminated when the first character of a C-string pointed at by one of the array of pointers is '\0'.

You should inspect this program carefully and understand it because the methods of single and double indirection that it shows are generally applicable for all cases in C++ where arrays of pointers are used.

```
/**********************************************************************
 *
 *     'array2d.cpp' -- Program to initialise a two-dimensional
 *                   character array and display its contents
 *
 **********************************************************************/
#include <iostream>
using namespace std;

int main()
{
    char *cptr[] = {"Signor Antonio, many a time and oft\n",
                    "on the Rialto, you have rated me\n",
                    "for my moneys and my usances.\n",
                    "Still have I borne it with a patient shrug,\n",
                    "for sufferance is the badge of all our tribe ",""};

    char **cpp;     // Pointer to array of pointers
    char reply[5];

    // Display all the strings using subscripts

    cout << endl << "Press RETURN to continue ";
    gets(reply);

    for (int i = 0; *cptr[i]; i++)
        cout << cptr[i];

    // Now do the same, with pointers

    cout << endl << "Press RETURN to continue ";
    gets(reply);

    for (cpp = cptr; **cpp; cpp++)
        cout << *cpp;
}
```

The output of array2d.cpp is this

```
Press RETURN to continue
Signor Antonio, many a time and oft
on the Rialto, you have rated me
for my moneys and my usances
Still have I borne it with a patient shrug,
for sufferance is the badge of all our tribe
Press RETURN to continue
Signor Antonio, many a time and oft
on the Rialto, you have rated me
for my moneys and my usances
Still have I borne it with a patient shrug,
for sufferance is the badge of all our tribe
```

Command line arguments

In all examples presented in earlier chapters, you have entered data via the input stream operator cin or one of the functions (e.g. cin.get) of the istream class. The main function has never been supplied with arguments.

You can make the main function take arguments so that the user can enter a command at the shell level of the operating system. The DOS copy command

```
C:\> copy file1 file2
```

shows a C++ program in operation. You are not, however, prompted for the file names. As you would expect, you enter them on the command line instead.

To set up command-line arguments in your C++ program, you use the special arguments argc and argv in the main function header. The main function header with command-line arguments looks like this:

```
int main(int argc, char *argv[])
```

argc is an integer value that holds the number of arguments on the command line. Its minimum value is 1, because the name of the program qualifies as an argument. In the copy example above, the value of argc is 3.

argv is a pointer to an array of character pointers. Each of the character pointers in the array points to a C string. Each of the C strings is a single command line argument. Again considering the copy example:

```
argv[0]     points to     "copy"
argv[1]     points to     "file1"
argv[2]     points to     "file2"
```

argv[argc] is always a null pointer. In the copy example, argc has the value 3, which is one more than the number of arguments, counting from zero.

The empty brackets [] of argv indicate that it is an array of undetermined length. Its actual length is established at runtime, when it is initialised with the command-line arguments entered by the user.

You could also write the main header as

```
int main (int argc, char **argv)
```

In the program code, *argv could be used in place of argv[0], *++argv in place of argv[1], *++argv instead of argv[2], and so on.

In this case, keeping track of pointers is less convenient than using subscripts. Any performance overhead caused by subscripting a three-element array is negligible, which is why double indirection on command-line arguments is often not used.

There follows a minimal example of a complete program, cmdarg.cpp, that uses command line arguments. It doesn't do anything other than accept the command line and, using various techniques, display the individual arguments. Here it is:

```
/*******************************************************************
 *
 *     'cmdarg.cpp' — Demonstrate use of command-line arguments
 *
 *******************************************************************/
#include <iostream>
using namespace std;

#include <cstdlib>

int main(int argc, char *argv[])
{
    FILE *inp, *outp;
    char **argvp = argv + 1;

    if (argc != 3)
    {
        cout << "Program " << argv[0] << " usage: "
            << argv[0] << "<f1> <f2> " << endl;
        exit(0);
    }

    cout << "Command line entered: " << argv[0]
        << " " << *argvp << " " << argv[2] << endl;
}
```

cmdarg.cpp expects a command-line something like this:

```
cmdarg argtext1 argtext2
```

There must be three arguments in total, including the program's name. Otherwise the "Usage" message is displayed and program execution stops with the exit library function call. Assuming that three arguments are specified, then all three are displayed by the cout that follows. The program name and the third argument (argtext2) are displayed using subscripted references to argv. The program displays the first argument (argtext1) using a doubly indirected pointer. argvp is a pointer to pointer and is used to be the same as the argument pointer argv. If argvp is the same as argv and therefore points to the string "cmdarg", then argvp + 1 points to the string "argtext1".

If you run the program without arguments, the screen display will be similar to this:

```
Program C:\CMDARG.EXE. Usage: C:\CMDARG.EXE <f1> <f2>
```

With the proper number of arguments, you get the following:

```
Command line entered: C:\CMDARG.EXE argtext1 argtext2
```

Pointers to functions

Use of pointers to functions is one of the aspects of C++ syntax that most intimidates novice (and not-so-novice?) C++ programmers. In fact, function pointers are no more than a logical completion of the general pointer syntax.

Functions are not variables, but you can define pointers to them, store such pointers in arrays and pass them as arguments between functions.

Function pointers are typically used in specific classes of application:

- where a function's identity is to be supplied as an argument to another function
- where outside events determine which of many functions is to be called next. In such cases an array of pointers to functions is often used to control function calls.

A pointer to a function contains the internal memory address of the entry point of that function. The address of the function is obtained using only the function's name.

Here is how to define a pointer to a function:

```
int (*fptr)();
```

fptr is a pointer to a function returning an int. Note that all the parentheses here are necessary. For example,

```
int *fptr()
```

is not a pointer to a function, but the definition of a function returning a pointer to an int.

Simple example of a function pointer

The program drawline.cpp is a simple example using pointers to functions.

```
#include <iostream>
using namespace std;

void drawline(int)

int main()
{
    //    Define a pointer to a function with
    //    an 'int' as a parameter

    void (*fptr)(int len);
    //    Assign a function address to the pointer

    //    All the following assignments are
    //    good but some compilers reject the first two
    //    (*fptr) = drawline;
    //    (*fptr) = &drawline;
    //    fptr = drawline;
    //    fptr = &drawline

    fptr = drawline;

    //    (*fptr)(50); is OK for
    //    the function call also
    fptr(50);
}

void drawline(int len)
{
    while (len > 0)
    {
        cout << "-";
        len--;
    }
    cout << endl;
}
```

The program draws a horizontal line at the bottom of the screen display

Use of the function pointer is not necessary; you could as easily call the function drawline explicitly. fptr is defined as a function pointer. The name of the function, drawline, is the address of that function. It is assigned to fptr, which is then called as a function name exactly as drawline could be.

The function call using the pointer may alternatively be made with dereferencing syntax

```
(*fptr)(50);
```

Use in real C++ programs of function pointers is usually much more complex than this. However, the function pointer syntax of drawline.cpp is the basis of all usages of pointers to functions. In this *Made Simple* text, I'm not going to present further examples of programs using function pointers. If you want to find out more, I would again advise you to refer to the *C++ Users Handbook*.

Exercises

1. Write a program, **llstruct.cpp**, that implements the linked-nodes program described in the first section of this chapter.
2. Write a program, **dynstruc.cpp**, that implements the linked-nodes using dynamic memory allocation.
3. Write a program, **list.cpp**, that implements a full linked-list program using a loop to take repeated user input, and dynamic memory allocation to create each successive list element.

8 Classes

The class construct

The C++ class construct is a generalisation of the structure, found in its original and simplest form in the C language. In C, the struct is an aggregation of necessarily data members; in C++ the struct may additionally have function members. The C++ class and struct are the same except that the members of the class are by default of private (restricted) access while those of the structure are public.

Let's look at an example of a class, date

```
class date
{
private
    int dd
    int mm.
    int yy
public
    void get_data():
    int  validate():
    int  find day():
    void disp_day(int):
}
```

You can see that some of the members of date are private and some are public. In hierarchies of derived classes (more on this in Chapter 10), you can also use the protected keyword. The *access control* keywords private, public and protected may appear anywhere, in any order, between the curly braces. A public member of a class (very often a member function) can be accessed by external (client) code that is in no way part of the class. A private class member, on the other hand, can only be used by code defined in a member function of the same class.

If none of the access-control keywords is used in a class declaration, then all its members are by default private. In a struct declaration, omission of all these keywords means that all members of the structure are by default public. This is the only difference between the class and struct constructs in C++.

To be useful, a class must have some accessible (usually public) functions that may be called from client code to access indirectly the private data and function members of the class. The data members of the date class are private; the member functions are callable by any client code for which they are in scope.

If you don't insert the keyword private before the data members, they become private anyway, as that's the default access level for class members. If you left out all access specifiers, then all the members of the class would be by default private and the class would effectively be inaccessible and useless. Although class members are private by default, the preferred form is to use private explicitly.

You should note that my placement of all the private class members before the public ones is only my preference; the public members could precede the private ones, and private and public declarations can be intermixed.

You define an instance of the date class with

```
date day;
```

and an array of class instances like this

```
date day_arr[20];
```

You can use the terms *class object* and *class variable* synonymously with *class instance*.

Here's how to define and initialise a pointer to the class instance day:

```
date *dptr = &day;
```

Use of pointers with classes and class members is covered later in this chapter.

You can't initialise a class or structure with an initialiser list in the way structures are initialised in C:

```
date day = {22,08,02};
```

Classes and structures should be initialised with constructor functions, which you can see in Chapter 9.

Members of a class are in scope for the whole outer block of the class declaration – between the curly braces. This means that member functions can directly access the other function members. If you want to get at a member function of the class variable day from client code, you have to do it by qualifying it with the class object day and the member-of (dot) operator:

```
day.get_data();
```

Within the definition of get_data, you can access the other members *without* qualification.

```
void date::get_data()
{
    char c;

    cout << "Enter the day number: ";
    cin >> dd;
    cout << "Enter the month number: ";
    cin >> mm;
    cout << "Enter the year number: ";
    cin >> yy;
    // Flush the last RETURN from the input stream

    c = cin.get();
}
```

Example: The date class

Here's a somewhat cut-down version of the date class. It's organised in three files: the header file dates.h, the function program file datefunc.cpp, and the main program file dates.cpp, which calls the functions declared as part of the class date. First, we have the dates.h header file:

```
// dates.h
extern const int MINYY;
extern const int MAXYY;
extern const int MINMM;
extern const int MAXMM;
extern const int MINDD;
extern const int MAXDD;
extern const int MINFEB;
extern const int MAXFEB;
extern const int TRUE;
extern const int FALSE;
class date
{
private:
     int dd;
     int mm;
     int yy;
public:
     void get_data();   // read input date
     int  validate();       // check date for correctness
};
```

dates.h declares a number of symbolic constants used in date validation. It also declares a shortened version of the date class already introduced. The (now only two) member functions of date are defined in datefunc.cpp:

```
// datefunc.cpp
#include <iostream>
using namespace std;
#include "dates.h"
void date::get_data()
{
     char c;
     cout << "Enter the day number: ";
     cin >> dd;
     cout << "Enter the month number: ";
     cin >> mm;
     cout << "Enter the (4-digit) year number: ";
     cin >> yy;
// Flush last RETURN from the input stream
     c = cin.get();
}
int date::validate()
{
     // Validate the date entered according to
     // the well-known rules
     if ((yy < MINYY) || (yy > MAXYY))
          return(FALSE);
     if ((mm < MINMM) || (mm > MAXMM))
          return(FALSE);
     if ((dd < MINDD) || (dd > MAXDD))
          return(FALSE);
     if  ((mm==4)||(mm==6)||(mm==9)|| (mm==11))
          if (dd > (MAXDD - 1))
               return(FALSE);
     // If the month is February and the year is divisible evenly by 4,
     //  we have a leap year

     if (mm == 2)
     {
          if (dd > MAXFEB)
               return(FALSE);
          if ((yy % 4) != 0)
               if (dd > MINFEB)
                    return(FALSE);
     }
     // If this point is reached, we return a valid date indicator
     return(TRUE);
}
```

The date::get_data function does what its name suggests: it prompts the user for input of three numbers constituting a date. The form date::get_data uses the binary scope resolution operator to specify that I'm referring to the get_data member function of the class date and not some other get_data function. The function date::validate checks the three numbers for correctness as a date and returns TRUE or FALSE accordingly. The function only operates on the years 1901 to 2099.

Finally, here's the dates.cpp program file. It contains definitions of the symbolic constants declared in dates.h (use of symbolic constants in this way is generally considered superior to and more "C++ish" than preprocessor definitions), followed by a main function which calls the date member functions.

```
// dates.cpp
#include <iostream>
using namespace std;

#include "dates.h"

//   define global symbolic constants
const int MINYY       = 1901;
const int MAXYY       = 2099;
const int MINMM       = 1;
const int MAXMM       = 12;
const int MINDD       = 1;
const int MAXDD       = 31;
const int MINFEB      = 28;
const int MAXFEB      = 29;
const int TRUE        = 1;
const int FALSE       = 0;

int main()
{
    int c;
    date datein;

    // Stop user data-input when 'q'-RETURN
    // is entered

    cout << "Press RETURN to continue, 'q'-RETURN to quit: ";
    while (c = cin.get(), c != 'q' && c != EOF)
    {
        datein.get_data();
        if ((datein.validate()) == FALSE)
            cout << "Invalid date entered\n";
        else
            cout << "Date entered is OK\n";
        cout << "Press RETURN to continue, ";
        cout << "'q'-RETURN to quit: ";
    }
}
```

The header file iostream is included in both datefunc.cpp and dates.cpp. It contains, among other things, all declarations necessary to allow use of the input and output streams cin and cout, as well as the stream I/O function get. dates.h is also included in both files, making the symbolic constants and the date declaration visible throughout the program.

Class members

Data members

You declare data members of a class within the class in the same way as ordinary (non-class-member) data objects. The class cust_acc

```
class cust_acc
{
private
    float bal;
    int acc_num
public
    // member functions
};
```

can equally well be written

```
class cust_acc
{
private
    float bal; int acc_num;
public:
    // member functions
};
```

Static data members

You can't qualify declaration of class data members with any of auto, register or extern. If you declare a data member static, only one copy of that data object is allocated by the compiler in memory, regardless of how many instances of the class are defined. A static member therefore acts as a global variable within the scope of a class and might reasonably be used as a global flag or counter variable. Here's a simple example:

```
#include <iostream>

using namespace std;

class run_total
{
private
    static int accum
public:
    void increment() { accum++; }
    void pr_total()
    {
        cout << "Accum: " << accum << "\n";
    }
};
//  definition of static member
```

```
int run_total::accum = 0;

int main()
{
    run_total total1, total2;

    total1.increment();
    total1.pr_total();
    total2.increment();
    total2.pr_total();
}
```

In this program, we define two instances of the class run_total, total1 and total2. After the first call to increment, the value of accum is 1. After the second call to increment – albeit with a different class instance – the value of accum becomes 2.

Static data members should be defined outside the class declaration. This is the reason for inclusion of the line

```
int run_total::accum = 0;
```

in global scope (outside all functions and classes). Static data members must not be initialised in this way more than once in the program.

Static data members of a class exist independently of the existence of any instances of that class; space for them is allocated at compile-time. Nevertheless, a static data member declared in this way is not a runtime definition. Additionally, although compilers often implicitly initialise such members to zero and allow their use without an explicit definition, the language specification doesn't guarantee that they will.

Nested class declarations

You can declare classes (including structures) as data members of a class. The declaration of the member class must already have been encountered by the compiler.

```
class cust_details
{
private:
    char accountName
    int age;
public:
    // 'cust_details' member functions
}
class cust_acc
{
private:
    float bal;
```

```
        int acc_num

    public
        cust_details resume
        // 'cust_acc' member functions
    }
```

Here, the class cust_details is declared before an object of its type is defined in the cust_acc class

Function members

You can specify all the code of a member function, or just its prototype, within a class declaration. In addition, you have the option of using either of two function specifiers, inline and virtual.

If you specify a function inline as part of its declaration, the compiler is requested to expand the body of the function into the program code at the point of its call. In this way, it is treated in much the same way as a preprocessor macro: the function is expanded inline and the overhead of the function call is eliminated. If a class member function is *defined as part of its declaration*, it is *implicitly* inline

```
    class cust_acc
    {
    private
        float bal;
        int acc_num
    public
        void zero_bal() { bal = 0.0; }
        //   Other member functions here
    };
```

Prefixing the inline specifier to the function definition within cust_acc is unnecessary and makes no difference to the definition of zero_bal. You can regard the function zero_bal as shown as implicitly inline

You don't have to include a function's entire definition in a class declaration for the function to be inline. You can declare a member function inline and define it later

```
    class cust_acc
    {
    private
        float bal
        int cust_acc;
    public.

            inline void balance();
```

```
}

//  function definition
void cust_acc::balance()
{
```

A particular type of implicitly inline function called the *access function* is very useful for hiding of private member data objects. For example

```
class cust_acc
{
private:
    float bal;
    int acc_num;
public:
    int isOverdrawn() { return(bal < 0.0); }
    //  Other member functions here
}
```

Here, the boolean value of the equality test bal < 0.0 is returned by isOverdrawn. With this mechanism, you don't have to access the variable bal to check the customer's creditworthiness; you can instead do it with the function call

```
    cust_acc a1;


if (a1.isOverdrawn())
        //  don't give her the money
```

A short function like this is particularly suitable for inline specification. Access functions are very common. They make it unnecessary for client code directly to access data members. The data hiding that results allows you to change the class definition while having no effect on the operation of the client code.

I deal with virtual functions, declared with the function specifier virtual, as seen in Chapter 10.

Ordinary member functions are those not specified inline or virtual and which are defined outside the class declaration. Their function headers must contain the scope resolution operator, as in the case of balance from the cust_acc class:

```
void cust_acc::balance()
```

You can't declare a class data member twice in the same class. You *can* declare a member *function* twice in the same class but only if the two declarations have different argument lists. You can see the rules for declaration of overloaded functions in Chapter 9. Lastly, you're not allowed to declare a member data object and a member function with the same names.

Static member functions

A static member function is allowed access only to the static members of its class, unless it uses a class object with one of the operators '.' or '->' to gain access. To illustrate, here is a modified version of run_total from earlier in this section:

```
#include <iostream>
using namespace std;

class run_total
{
private:
    static int accum;
public:
    static void increment() { accum++; }
    void pr_total()
    {
        cout << "Accum: " << accum << endl;
    }
};
int run_total::accum = 0;

int main()
{
    run_total total1, total2;

    total1.increment();
    total1.pr_total();
    total2.increment();
    total2.pr_total();
}
```

Now, as well as accum, the function increment has been declared static and can still access accum. If, however, the static keyword is removed from the declaration of accum, a compilation error results. The function increment can access a non-static data member of the same class by using, in this case, a class object to qualify accum:

```
#include <iostream>
using namespace std;

class run_total
{
private:
    int accum; // non-static
public:
    static void increment(run_total& inst)
    {
        inst.accum++;  // this usage OK
    }
```

```
        void pr_total()
        {
            cout << "Accum: " << accum << endl;
        }
    };

    int main()
    {
        run_total total1, total2;

        total1.increment(total1);
        total1.pr_total();
        total2.increment(total2);
        total2.pr_total();
    }
```

In the examples above, you should note that the static member function increment can be used without reference to instances of the class run_total:

```
int main()
{
    run_total total1, total2;

    run_total::increment();
    total1.pr_total();
    run_total::increment();
    total2.pr_total();
}
```

Here, only access to the non-static function pr_total must be controlled by the class instances total1 and total2.

Example: Using static class members

As a more practical example of a case in which static class members might be used, here's the bank-account example from Chapter 1 reworked so that the account number is no longer prompted for in the setup function. Instead, each time you create an account instance, the next available number is peeled off. In summary, you need a variable global to all cust_acc class instances to hold information logically common to them all.

```
    // accounts.h
    class cust_acc
    {
    private:
        float bal;
        static int acc_num;
```

```
     int my_acc_num;
public:
     void setup();
     void lodge(float);
     void withdraw(float);
     void balance();
};

//   Program file accfunc.cpp -- defines
//   cust_acc member functions.
#include <iostream>
using namespace std;
#include "accounts.h"
//
//   Only setup function has changed
//
void cust_acc::setup()
{
     my_acc_num = acc_num++;
     cout << "Enter opening balance for account "
          << my_acc_num << ": ";
     cin >> bal;
     cout << "Customer account " << my_acc_num
                << " created with balance " << bal << endl;
}
// account.cpp
#include <iostream>
using namespace std;
 #include "accounts.h"
int cust_acc::acc_num = 1000;
int main()
{
     cust_acc a1;
     a1.setup();
     a1.lodge(250.00);
     a1.balance();
     a1.withdraw(500.00);
     a1.balance();
     cust_acc a2;
     a2.setup();
     a2.lodge(1000.00);
     a2.balance();
     a2.withdraw(300.00);
     a2.balance();
}
```

Friends

In a strict OOP world, only public member functions of a class are allowed direct access to the private member variables. Things are not that simple, however, and C++ provides the friend mechanism, which allows the rules to be bent.

A function may be specified within a class declaration and prefixed with the keyword friend. In such a case, the function is *not* a member of the class, but the function is allowed access to the private members of the class. Here is the cust_acc class containing a friend declaration:

```
 class cust_acc
{
private:
    float bal;
    static int acc_num;
    int my_acc_num;
public:
    void setup();
    void lodge(float);
    void withdraw(float);
    void balance();
    friend void enquiry();
}
```

The function enquiry is not a member of the class, but you can call it from anywhere else in the program and it nevertheless has full access to all members of cust_acc, even the private ones.

You're encouraged to be sparing in your use of friend declarations. Too many friends can be a bad thing. A case where friends are useful, even necessary, is that of operator overloading, of which more in the next chapter.

Class scope

As stated in Chapter 3, every C++ data object has local, function, global (or file), namespace or class scope. Scope defines the visibility of a data object. If it has global file scope, it is visible throughout the program file in which it is defined and is said to be global. If a data object has function scope, it is visible only within the function in which it is defined. Only goto labels have function scope. If a data object has local scope, its visibility is confined to the local enclosing compound statement.

A C++ class has its own scope. This means that a class member is directly visible only to member functions of the same class. Access to the class member is otherwise limited to cases where the member-of (.), pointer (->) and scope-resolution (::) operators are used with either the base class or a derived class. A data object declared as a friend of a class belongs to that class's scope.

Here is a modified example of the date class, which illustrates the different aspects of class scope:

```
class date
{
private:
    int dd;
    int mm;
    int yy;
public:
    void get_data();
        { cin >> dd >> mm >> yy; } // inline
    int  validate();
    int  find_day();
    void disp_day(int);
};
```

In client code, such as the main function, we define an instance of the class and a pointer to it:

```
date   day;
date   *dptr = &day;
```

For all of the four member functions, all other class members are in scope. Thus the code of the validate function might, if necessary, call the function disp_day, even though disp_day is declared later in the class than validate. Member function code may access other class members – data and function – directly, without using any prefixes to resolve scope.

To access function members from client code, you must use the member-of and pointer operators:

```
day.dd
dptr->dd
```

(although you could only do this if dd were not of private access). Likewise:

```
day.validate()
```

is equivalent to

```
dptr->validate()
```

If you don't use these prefixes, the members are out of scope for the client code and compilation errors result. The private class members are always out of scope for client code. You can only access them indirectly using member functions, for which they are in scope.

To see the effect of the scope resolution operator, let's look at a modified version of the run_total example from page 164.

```
#include <iostream>
using namespace std;
class run_total
{
private:
    static int accum;
public:
    static void increment() { accum++; }
    void pr_total()
    {
        cout << "Accum: " << accum << endl;
    }
};

int run_total::accum = 0;
int main()
{
    run_total total1, total2;
    int run_total = 11;
    run_total::increment();
    total1.pr_total();
    run_total::increment();
    total2.pr_total();
    cout << run_total << "\n";
}
```

Here, although the class name, run_total, is redefined as an integer in main, class scope is *resolved* in the calls to the static member function increment, by means of the binary scope resolution operator. The result of the program is

```
Accum: 1
Accum: 2
11
```

With derived and nested classes, use of the scope resolution operator is at times necessary to avoid ambiguity when accessing class members. Otherwise, avoid masking declarations in this way: it doesn't help program reliability or readability.

Nested class declarations

Where a class is declared in class scope the declaration is said to be nested one class is declared within another Declarations made in the nested class are not in scope for functions in the enclosing class and must be accessed according to the normal procedures Equally declarations in the enclosing class are not in scope for functions declared in the nested class.

Here are the relevant parts of an example program again based on the date class that shows use of nested classes.

```
 // file dates.h' contains

    // nested classes 'date' and 'curr_time'

    class date
    {
    private
        int dd
        int mm
        int yy
    public
        class curr time
        {
        private
            int hr
            int min
            int sec.
        public
            void correct_time();
        };
        void get_data();
        int validate();
        int find_day();
        void disp day(int).
    }
```

The nested class curr time is added to the date class curr time is declared and an instance of it defined within date In this case the function correct_time is used to reset the data members of class curr timo, probably by calling library functions declared in the standard header file ctime.

The calling sequence for this function is

```
    date day


    day t correct_time();        // set correct time
```

To conform with the C++ scope rules, you must write the header of the correct_time function like this.

```
        void date: curr_time correct time()
```

The definition of an instance of the class date also defines an instance of curr_time because of the definition of t embedded in date. The members of a nested class are not in scope for those of the enclosing class, to qualify the function header of correct_time only with the scope resolution date would cause the function correct_time to be out of scope even though it is a member of a class nested within date

ISO C++ has introduced an extension allowing forward declaration of nested classes. In the example above showing the curr_time class nested within date, a forward declaration of curr_time can instead be used

```
 class date

    {
    private:
         int dd;
         int mm;
         int yy;
    public:
         class curr_time;
         curr_time t;
         void get_data();
         int  validate();
         int  find_day();
         void disp_day(int);
    }

    class curr_time
    {
    private:
         int hr;
         int min;
         int sec;
    public:
         void correct_time();
    }
```

The first display command reads

```
cout << xptr->*dptr << " " << yptr->*dptr << "\n"
```

Member function pointers

Use of the specialised pointer-to-class-member syntax may be desirable in all cases where class members are to be accessed using pointers but you have to use it where a member function is to be called with a pointer. You can't access member functions of a class using conventional function pointers. For example, a conventional pointer to function returning integer:

```
int (*fptr)();
```

can't be used to point to a member function of a class, even if that function exactly matches the pointer definition in signature.

Consider the coord class with a function member:

```
class coord
{
private:
    int x_coord;
    int y_coord;
public:
    int locate_coords();
};
```

You can't use a conventional function pointer to point to the function locate_coords. Instead, we define a pointer to member function:

```
int (coord::*mem_fn_ptr)();
```

assign a function address to it like this:

```
mem_fn_ptr = coord::locate_coords;
```

and call it:

```
mem_fn_ptr();
```

Use of the member-pointer operators provides better control than using ordinary pointers to point to members and less likelihood of pointers being used for unintended purposes. Unfortunately, the syntax is somewhat complicated. This may encourage programmers to stick where they can with traditional pointers and (as even C++ programmers are inclined to do) avoid function pointers altogether.

Classes as function arguments

Pointers to classes are sometimes used where class instances are being passed as arguments to functions. In C++, you usually use reference declarations instead to achieve the same purpose.

```
#include <iostream>
using namespace std;
class fraction
{
public:
    double f;
    double g;
};
// Function prototype
void change_class(fraction&);

int main()
{
    fraction x, y;
    double fraction::*dptr;
    x.f = 1.1;
    y.f = 2.2;
    x.g = 3.3;
    y.g = 4.4;
    change_class(x);

    dptr = &fraction::f;
    cout << x.*dptr << " " << y.*dptr << endl;
    dptr = &fraction::g;
    cout << x.*dptr << " " << y.*dptr << endl;
}

void change_class(fraction& xptr)
{
    xptr.f = 5.5;
    xptr.g = 6.6;
}
```

When you run the program, you get this result:

```
5.5 2.2
6.6 4.4
```

Depending on how it implements the C++ language, the compiler may replace the reference code with pointer referencing and dereferencing syntax. In any event, you're saved from having to do it. The reference (a trailing &) is only referred to in the change_class prototype and function header. In change_class, the class members are accessed as if the function had been called by value, with the argument x.

Reference declarations qualified by const are strongly recommended if you don't want a called function to change the value of its parameter.

```
void dont_change_class(const fraction& xptr)
{
    // compile error if x members changed
}
```

You can suffix a function itself with const.

```
class fraction
{
    //
public:
    dont_change_members() const
    {
        //
    }
}
```

A const function generates compilation errors if it attempts to change the value of members of the class object with which the function has been called. The const suffix only has meaning for class member functions.

The this pointer

Every member function of a class has an implicitly defined constant pointer called this. The type of this is the type of the class of which the function is member. It's initialised, when a member function is called, to the address of the class instance for which the function was called.

Here's a representative example of the use of this:

```
#include <iostream>
using namespace std;
class coord
{
private:
    int x_coord, y_coord;
public:
    void set_coords(int x_init, int y_init)
    {
        x_coord = x_init;
        y_coord = y_init;
    }
    void change_coords(int, int);
    void display_coords()
    {
        cout << "Coordinates: " << x_coord << " " << y_coord << endl;
    }
}
```

```
int main()
{
    coord c1;

    c1.set_coords(5, 10);

    cout << "Original C1" << "\n";
    c1.display_coords();
    c1.change_coords(15, 20);

    cout << "Changed C1" << "\n";
    c1.display_coords();
}

void coord::change_coords(int x_chg, int y_chg)
{
    coord c2;

    c2.set_coords(x_chg, y_chg);

    cout << "Display C2" << endl;
    c2.display_coords();

    *this = c2;
}
```

The program produces these results:

```
Original C1
Coordinates: 5 10
Display C2
Coordinates: 15 20
Changed C1
Coordinates: 15 20
```

The this pointer is useful when you want during execution of a class member function to get a 'handle' on the class object used to call the function. Because in a member function the class variable with which the function was called is out of scope, the this pointer is provided as that 'handle'.

Whether in class member functions the this pointer is explicitly used or not, the C++ compiler accesses all class members using an implicit this pointer.

Static member functions do not have this pointers. There is only one instance of a static member function for a class, so use of this does not make much sense. Any attempt to use this in a static member function causes a compilation error. Static member functions may otherwise be accessed by means of pointers using the same syntax as non-static member functions.

Exercises

1 Given the abstract object clock, identify attributes of the class clock. Declare clock as a C++ class. Ensuring that the class contains at least one member function, write down the definitions (as they would be written in a separate .cpp file) of each of the functions and show how they would be called from an external function such as main.

2 Given the class declaration

```
class policy
{
private:
    char name[30];
    char address[50];
    char polno[8];
    double ins_value;
    double premium;
public:
    void pol_open();
    void pol_close();
    void renew();
    bool claim(double);
};
```

What is wrong with this definition of an instance

```
policy jsmith = {"J. Smith","Valley Road","12345678",1000.00,100.00};
```

Why? How should it be done?

3 Change the policy class as it appears in 2 above so that each instance of the class takes its policy number from a static member glob_polno. Initialise glob_polno appropriately to a value of 10000. Create at least two instances of policy and demonstrate that they have been set up with different policy numbers.

9 Class services

Introduction

This chapter describes the facilities provided by C++ to allow you to work with class instances without having to know what's inside. For example, suppose we had an instance A1 of the bank-account class cust_acc. We might want to transfer the account to A2. The instance-level assignment

```
A2 = A1
```

is much nicer and more intuitive than 'reaching inside' the instances and copying the members one by one. We might want to create a new account A3 and set it up initially with the contents of A2

```
cust_acc A3(A2)
```

and for this we need a *copy constructor*. We might want, at the instance level, to add money (pounds, dollars, euros, whatever) to the account. It would be attractive to be able to write:

```
A3++;
```

to add one pound or

```
A3+=5;
```

to add five euros. You're looking here at two cases of *overloaded operators*.

This chapter, in essence, concerns itself with constructors and overloaded operators. It describes how you can use these two C++ language facilities to work with classes at the instance level, without having to be aware of the internals. Step into the shoes of the C++ programmer who *uses* classes defined and implemented by others, and these high-level facilities make life a whole lot simpler. Put on your class designer's hat, on the other hand, and you find that you have to know how to use these facilities in order to hide the details from the programmers who will be using *your* classes.

Constructors, destructors, overloaded operators and, especially, their side-effects are not simple. But this book is a *Made Simple* so what follows is the 'short path': a straightforward presentation of the essentials. If you want to get into the 'dark corners' — and, believe me, there are plenty of them in this area — look beyond this book to my other publication, the *C++ Users Handbook*, or the *C++ Programming Language (3rd edn)* by Stroustrup.

Constructors and destructors

When you define a variable in C++ you have no automatic mechanism for ensuring that the variable is set to some reasonable value when it is created or that the variable is tidied up (for example its memory deallocated) immediately before it is destroyed

Constructor and destructor functions are introduced in C++ for this purpose Constructors and destructors are (and must be) class member functions that have the same name as the class of which they are a part In the case of the destructor, the name is prefixed with a tilde '~'

Here s an abstract example that has the single virtue of being short

```
class newclass
{
private

    //   private data members defined here

public
    newclass()              // constructor function
    {
        // initialising statements here
        cout << "Constructing.. " << endl
    }

    //   other public members defined here

    ~newclass() //destructor function
    {
        // un-initialising statements here
        cout << "Destructing...." << endl.
    }
}
```

Here the constructor function **newclass** is defined as a public member function of the class of the same name You don t have to make a constructor **public**, it can be **private** or **protected** and it can be anywhere in the list of member functions Similarly the destructor function **~newclass** need not be declared **public** and may be declared anywhere among other declarations.

All the same you should note that constructors and destructors are usually declared with public access If private they are more difficult to use because access to them is restricted to member functions of the same class

When you define an instance of **newclass**

```
newclass  nc:
```

an instance **nc** of the class **newclass** is defined and the initialising statements in the body of the **newclass** constructor function are executed When **nc** goes out of scope.

the destructor function ~newclass is implicitly called and its uninitialising statements do suitable tidying-up operations, which usually include returning storage to the system's free list.

A destructor is almost always called implicitly in this way. You will very rarely, if ever, explicitly call a destructor function.

You should note that constructor functions do not create class objects, nor do destructor functions destroy them. A class object is created when you define it; creation is immediately followed by execution of the body of the constructor function. Conversely, when a class object goes out of scope, its destructor function is executed and only then is the object destroyed.

Let's look at how constructor and destructor functions are called. First, here is the declaration of newclass:

```
class newclass
{
private:
    int a, b, c;
public:
    newclass()
    {
        a = b = c = 0;
        cout << "Constructing..." << endl;
    }
    ~newclass()
    {
        cout << "Destructing..." << endl;
    }
};
```

Next, here are the functions that use newclass:

```
int main()
{
    newfunc();
}

void newfunc()
{
    newclass nc1;
    {
        cout << "Defining nc2..." << endl;
        newclass nc2;

    }
    cout << "Out of scope of nc2..." << endl;
}
```

newclass has three data members, all integers, and two member functions, its constructor and destructor. The constructor sets the three integers to zero and displays a message. The destructor simply displays a message. In the function newfunc, two instances, nc1 and nc2, of newclass are defined. The definitions call the constructor; when the definitions go out of scope, the destructor is implicitly called.

The displayed output of the program is this:

```
Constructing...
Defining nc2
Constructing...
Destructing...
Out of scope of nc2...
Destructing...
```

Constructor and destructor functions must not have return types, not even void. They may contain return statements but when return is used in this way it must have no operands. Only return; is valid. Constructors may take parameters; destructors must not, although void may be specified as a destructor argument list.

Simple constructor example

Here, once again using the bank-account class example, is a simple use of constructors. In previous declarations of cust_acc (see Chapters 1 and 8) we've used the member function setup to initialise the data members. This means that after defining an instance of cust_acc, you must remember to call setup to do the initialisation. In the next example, we replace this two-step procedure with a constructor. A destructor is also included, in this case mainly for illustration. The reworked cust_acc class is declared in the accounts.h header file:

```
class cust_acc
{
private:
     float bal;
     int acc_num;
public:
     cust_acc();
     void lodge(float);
     void withdraw(float);
     void balance();
     ~cust_acc()
     {
          cout << "Account " << acc_num
               << " closed" << endl;
     }
};
```

The program file accfunc.cpp contains the definitions of the class member functions other than the destructor. These definitions are unchanged from the examples shown in Chapter [illegible], except that setup is replaced by a constructor and a destructor function is added.

```
//   'accfunc.cpp'
#include <iostream>
using namespace std;
#include "accounts.h"
//
//   customer_account member functions
//
cust_acc::cust_acc()
{
    cout << "Enter number of account to be opened: ";
    cin >> acc_num;
    cout << "Enter initial balance: ";
    cin >> bal;
    cout << "Customer account " << acc_num
                << " created with balance " << bal << endl;
}

void cust_acc::lodge(float lodgement)
{
    bal += lodgement;
    cout << "Lodgement of " << lodgement << " accepted" << endl;
}

void cust_acc::withdraw(float with)
{
    if (bal > with)
    {
        bal -= with;
        cout << "Withdrawal of " << with << " granted" << endl;
        return;
    }
    cout << "Insufficient balance for withdrawal of "
          << with << endl;
    cout << "Withdrawal of " << bal << " granted" << endl;
    bal      = (float)0;
}

void cust_acc::balance()
{
    cout << "Balance of account is " << bal << endl;
}
```

Constructors taking parameters

Constructors are functions and can take parameters like any other function. Here is a simple example, in the program cstarg1.cpp, of a class that uses a constructor function taking parameters.

```
#include <iostream>
using namespace std;
class coord
{
private:
    int x_coord, y_coord;
public:
    coord(int x, int y)
    {
        x_coord = x;
        y_coord = y;
    }
    void print()
    {
        cout << x_coord << endl;
        cout << y_coord << endl;
    }
};
int main()
{
    coord point1 = coord(5,10);
    point1.print();
    //  coord point2;              // illegal
    coord point3(15,20);           // abbreviation
    point3.print();
}
```

This program illustrates several aspects of constructor parameter syntax. The constructor function coord – defined in full in the class and therefore implicitly inline – takes two integer parameters. From the code in main, you can see two ways of calling coord. The first

    coord point1 = coord(5,10);

is the full version: the function coord is called with the arguments 5 and 10 and the result of this function – the variables x_coord and y_coord set to 5 and 10 respectively – are assigned to point1, which is an instance of coord. The second constructor calling sequence

    coord point3(15,20)

is an abbreviation equivalent to the definition and initialisation of point1 above. You will usually use this abbreviated form of definition and constructor call in preference to the full version.

The simple definition in the last section

```
        cust_acc a1,
```

invokes the *default constructor* cust_acc::cust_acc(), which take no parameters. Even if you declare a class without any constructors explicitly included, the compiler includes such a default constructor. This is why the definition

```
        cust_acc a1,
```

works even where the class does not contain any explicit constructors. If you specify a default constructor yourself, that overrides the compiler-supplied default constructor. If you do not specify a default constructor but include a constructor that takes parameters, then the class *has no default constructor at all*. This is why, in cstarg1.cpp, the (commented out) definition of point2 is illegal: it tries to invoke the default constructor which, because of the presence of the two-parameter constructor, is absent. In this case, the correct constructor forms are the definitions of point1 and point3. The result of the program is simple

```
    5
    10
    15
    20
```

There is no destructor function in the class coord. You'll use destructors most often when a member function – usually the constructor – performs dynamic allocation of memory that should be freed at or before the end of program execution. In this case, no memory is dynamically allocated. You'll see constructors with dynamic allocation later in this chapter.

Example: Constructors taking parameters

Here's a more substantial example of use of constructors – with and without parameters – in the familiar cust_acc class. First the header

```
//   'accounts.h
class cust_acc
{
private
     float bal,
     int acc_num,
public
     cust_acc(),
     cust_acc(int, float);          // overloaded constructor
     void lodge(float).
     void withdraw(float).
     void balance();
}
```

```
//   'accfunc.cpp'
#include <iostream>
using namespace std;

#include "accounts.h"

cust_acc::cust_acc()
{
    cout << "Enter number of account to be opened: "
    cin >> acc_num
    cout << "Enter initial balance: ";
    cin >> bal;
    cout << "Customer account " << acc_num
            << " created with balance " << bal << endl;
}

cust_acc::cust_acc(int num_init, float bal_init)
{
    acc_num = num_init;
    bal = bal_init;
    cout << "Customer account " << acc_num
            << " created with balance " << bal << endl;
}

void cust_acc::lodge(float lodgement)
{
    bal += lodgement;
    cout << "Lodgement of " << lodgement << " accepted" << endl
}

void cust_acc::withdraw(float with)
{
    if (bal > with)
    {
        bal -= with
        cout << "Withdrawal of " << with << " granted" << endl;
        return
    }
    cout << "Insufficient balance for withdrawal of "  << with << endl;
    cout << "Withdrawal of " << bal << " granted" << endl;
    bal       = (float)0;
}
```

```
void cust_acc::balance()
{
    cout << "Balance of account is " << bal << endl;
}
```

And now the program

```
//  'accounts.cpp'
#include <iostream>
using namespace std;

#include "accounts.h"

int main()
{
    cust_acc a1;

    a1.lodge(250.00);
    a1.balance();
    a1.withdraw(500.00);
    a1.balance();

    cust_acc a2(12345, 1000.00);
    a2.balance();
    a2.withdraw(750.00);
    a2.balance();
}
```

The cust_acc class declaration now contains two constructor functions. The default constructor sets up objects of the cust_acc class using prompts, as you've already seen. The second constructor is an *overloaded constructor*. This is a special case of an overloaded function. The overloaded constructor

```
cust_acc::cust_acc(int num_init, float bal_init)
```

causes a new instance of the class cust_acc to be assigned the values specified by the two variables in the argument list.

In main, two instances, a1 and a2 of type cust_acc, are created. a1 is initialised by the constructor function cust_acc::cust_acc(), the default constructor. This prompts the user for input of the account number and opening balance, confirming that the account has been successfully opened. Definition of a2 causes the constructor with the argument list to be called. The variable members of a2 are assigned the argument values within that constructor. Here's the output of the program as I tested it:

```
Enter number of account to be opened. 12344
Enter initial balance 2000.00
Customer account 12344 created with balance 2000
Lodgement of 250 accepted
Balance of account is 2250
Withdrawal of 500 granted
Balance of account is 1750
Customer account 12345 created with balance 1000
Balance of account is 1000
Withdrawal of 750 granted
Balance of account is 250
```

Constructors and dynamic memory allocation

You can use constructors to initialise class objects for which memory has been dynamically allocated by the new operator.

```
#include <iostream>
using namespace std;

class coord
{
private:
    int x_coord, y_coord;
public:
    coord(int x, int y)
    {
        x_coord = x;
        y_coord = y;
    }
    void print()
    {
        cout << x_coord << endl;
        cout << y_coord << endl;
    }
}

int main()
{
    coord *p_coord;

    p_coord = new coord(5,10);

    p_coord->print();
}
```

Here, a new instance of the class type coord is allocated and its memory address assigned to the pointer p_coord. Additionally, the class's constructor function is called, initialising the data members of the class to the values 5 and 10.

A class object represented by an automatic variable is destroyed when that variable goes out of scope. On the other hand, a class object for which memory is dynamically allocated

```
class coord
{
    //
}

ptr = new coord
```

is persistent. When ptr goes out of scope, its destructor isn't called and the memory associated with ptr remains allocated. For the destructor to be invoked, you must explicitly deallocate the memory:

```
delete ptr;
```

which in turn causes the destructor to be implicitly called.

Function overloading in classes

You can use overloaded functions in defining classes, as well as in a procedural way, as seen in Chapter 3. Here's a class implementation of the squares program introduced in that chapter:

```
#include <iostream>
using namespace std;

class number
{
private:
    int num;
public:
    number() { num = 5; }       // constructor
    int Num() { return(num); }// access function

    // Function 'sqr_func' overloaded

    int sqr_func(int);
    float sqr_func(float);
    double sqr_func(double);
}

int main()
{
    number n;
    int i = n.Num();

    cout << n.sqr_func(i) << endl;
    cout << n.sqr_func( float(i) ) << endl;
    cout << n.sqr_func( (double)i ) << endl;
}

int number::sqr_func(int i)
{
    cout << "Returning int square: ";
    return(i * i);
}

float number::sqr_func(float f)
{
    cout << "Returning float square: ";
    return(f * f);
}
```

```
double number::sqr_func(double d)
{
    cout << "Returning double square: ";
    return(d * d);
}
```

The program uses a simple class, number, which defines one private integer member. This variable, num, is initialised by a simple constructor and its value retrieved in the function code using an access function. The value of num is assigned to the local variable i. The different instances of the overloaded function sqr_func are called depending on the type of i in the function calls.

The old-style typecast notation is used in the double call; the newer C++ equivalent is used for the float call. The results output by the program are:

```
Returning int square: 25
Returning float square: 25
Returning double square: 25
```

Operator overloading

Operator overloading is a special case of function overloading. You are allowed to assign additional meanings to most of the C++ *basic operators* like < (less than) and * (multiply). This means that you can define operators to do special processing not defined as part of C++.

The C++ basic operators that you may overload are:

!	~	+	-	*	&	/	%
<<	>>	<	<=	>	>=	==	!=
^	|	&&	||	+=	-=	*=	/=
%=	&=	^=	|=	<<=	>>=	,	->*
->	()	[]	=	++	--	new	delete

The operators on the last row in this table have some special characteristics when overloaded. For example, if you're a masochist, you can dispense with the memory management provided by your operating system and do it yourself by overloading the new operator. In one way or another, overloading the operators given on the last row can be regarded as advanced overloading. If you want all the ins and outs of this, have a look at the *C++ Users Handbook* or Stroustrup's *The C++ Programming Language*. This book, being a *Made Simple*, confines itself to non-advanced overloading and overloading the assignment, which is needed to provide a full range of class services.

You aren't allowed to overload these operators:

.   .*   ::   ?:

C++ doesn't allow new operators to be introduced by means of operator overloading. If you want to overload an operator, you must take it from the set of overloadable operators given above. For example, you might want to introduce an operator := to denote explicit assignment, as in Pascal, and to overload the equality operator == with the C++ assignment operator =. The introduction of := is illegal, the overloading of == with = is legal but confusing and undesirable.

To overload an operator, you must create a function named by the keyword operator immediately followed by the actual text of the operator to be overloaded. In the next example, we overload the addition operator +. Here's the + operator-overloading function:

```
char add_char::operator+(add_char& c2)
{
   // operator function code
}
```

This definition means that the overloaded-operator function named by operator+, which has a single class object parameter c2, carries out on c2 and the class of which operator+ is a member (add_char) a set of operations specified by the code in the body of the function.

The function name operator+ need not be a contiguous string. Any number of spaces may surround the operator symbol +.

Example: Overloading addition

Here's a simple example program, called add_char.cpp, that uses the class add_char and a member function which is the addition operator overloaded.

```
#include <iostream>
using namespace std;
class add_char
{
private:
    char c;
public:
    // constructor
    add_char(char c_in) { c = c_in; }
    // overloaded '+'
    char operator+(add_char& c2);
    char c_pr()            // access function
    {
        return(c);
    }
}
int main()
{
    add_char c1('g');
    add_char c2('h');
    char sum;

    sum = c1 + c2;
    cout << "'Sum' of " << c1.c_pr() << " and "
            << c2.c_pr() << " is " << sum << endl;
}
char add_char::operator+(add_char& c2)
{
    // add to the c1 character the alphabetic displacement of the c2 character
    //  This gives the 'sum' of the two characters.
    return(c + (c2.c - ('a' - 1)));
}
```

The purpose of the program is to perform alphabetic addition of characters using a + operator overloaded to do that special kind of addition. In the convention used by the program, c added to a is d, and h added to g is o. There is a mixed-type expression in the operator function that does the actual alphabetic addition.

The declaration of class add_char contains one private data member, c, of type char. It has three member functions: a constructor to initialise c to an alphabetic value, an access function to retrieve the value of c, and an overloaded-operator function giving a new meaning to the operator +.

In the main function, we define two instances of add_char, c1 and c2 and their data members initialised by the constructor to g and h respectively. We assign a local variable, sum, the result of the overloaded-operator function call

```
c1 + c2
```

The last statement in main displays that result

```
'Sum' of g and h is o
```

Now we look at the overloaded-operator function operator+. Here is its header:

```
char add_char::operator+(add_char& c2)
```

This specifies one parameter, which is a reference to the class object c2 corresponding to the operand on the right-hand side of the overloaded addition c1 + c2. In this addition, the operand c2 is the argument to the overloaded-operator function operator+. The operand being used as an argument to the operator+ function call doesn't have to have the same name as the function parameter. If the following class instances are defined and initialised:

```
add_char x1('c');
add_char x2('d');
```

it's OK to make x1 and x2 operands of the overloaded operator:

```
sum = x1 + x2;
```

The operand *x2* is then copied *through the reference* to the operator+ parameter c2. Use of the reference declaration add_char& c2 in the case of a simple class like add_char is not necessary, although it improves efficiency because copying a reference parameter imposes less overhead than copying a full class instance.

The overloaded-operator function operator+ is also passed an implicit this pointer to c1. The function can therefore directly access the data member c of c1.

In the return statement:

```
return(c + (c2.c - ('a' - 1)));
```

c is the private data member of c1, accessed using the implicit this pointer. Its contents are added arithmetically to those of c2.c, offset from the start of the alphabet. This last is an ordinary, not an overloaded, addition. You could write the return statement with the this pointer explicitly included:

```
return(this->c + (c2.c - ('a' - 1)));
```

Also, you can write the assignment to sum:

```
sum = c1.operator+(c2);
```

which may help you understand how the operator+ function receives an implicit this pointer referring to class object c1.

If you overload the assignment operator, =, the overloading function must be a member of a class. Functions overloading most other operators do not have to be class members but must take at least one argument that is a class object. This stipulation is designed to prevent a C++ basic operator being redefined unreasonably to operate on two non-class data objects. An example of unreasonable use would be to redefine the multiplication operator * to mean division when used with two integers.

Even with operator overloading, normal precedence and associativity of operators is unchanged. Thus, no matter how you might overload + and *, the expression

```
a + b * c
```

will always be evaluated as

```
a + (b * c)
```

You can't overload a basic operator that is strictly unary or binary to mean the opposite:

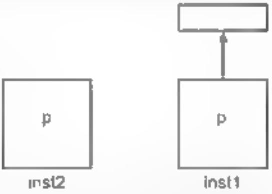
```
!x       //  '!' is always unary
17 % 6   //  '%' is always binary
```

In overloading operators you should try to mimic the purpose of the equivalent basic operator. The overloading of + in the program add_char.cpp is intuitive; + being overloaded to cause subtraction of characters would not be.

Overloading the assignment: Deep and shallow copy

Overloading the assignment operator presents a number of difficult underlying issues and yet you need to know about it. In this section, I try to give a not-too-detailed description of the mechanism and the side-effects of assignment that it is intended to overcome. The string class example later in this chapter gives every detail of the code required to implement the overloaded assignment.

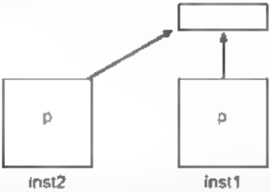
A class instance may be assigned to another of the same type. By default, *memberwise assignment* – a blind *bitwise* copy – is used. If the class objects contain pointer members, memory will become corrupted when those pointers are deallocated. Here's why: suppose we assign class instance inst1 to inst2 and that inst1 has a pointer member p that points to some dynamically-allocated memory. Before the copy, the objects can be shown with the diagram:



where the member, p, of inst1 points to an area of memory.

This is the situation after the assignment

```
inst2 = inst1
```



What we've done is a *shallow copy*: we've copied the pointers but not the memory they point to. Both pointers now point to the same area of memory. When inst1 and inst2 are about to go out of scope, calls to the destructor for both inst1 and inst2 will attempt (twice) to deallocate the same memory and a runtime error will result. We avoid this double-deallocation by overloading the assignment operator to copy the memory pointed to, not the pointers themselves. This is known as a *deep copy*.

Suppose we have a class called ptrclass, containing a pointer member, p. We have two instances of the class, inst1 and inst2. Here's how you'd overload the assignment operator so that memory doesn't get corrupted on assignment of one instance to the other:

```
class ptrclass
{
private:
    char *p;
public:
    //   public members here

    // overloaded assignment operator,
    // copies memory within instances
    ptrclass& operator=(ptrclass&);

    ~ptrclass() { delete p; }
};
```

The operator= function takes as its parameter a reference to the class instance on the right side of the overloaded assignment. When called, it's also implicitly passes a this pointer to the class instance on the left side of the assignment. It modifies that instance and returns a reference to it as the result of the assignment. Here's the skeleton of the operator= function.

```
// 'operator='

ptrclass& ptrclass::operator=(ptrclass& inst1)
{
    // Here, deep-copy the MEMORY AT inst1.ptr
    // to the MEMORY AT inst2.ptr, NOT
    // simply the pointer inst1.ptr to inst2.ptr

    return(*this);
}
```

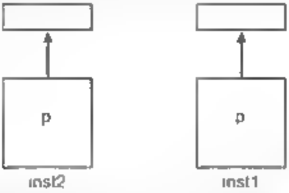
You do the overloaded assignment of the two instances like this:

```
inst2 = inst1;
```

which can also be written as:

```
inst2.operator=(inst1);
```

The operator= function is called with a reference to inst1 as an argument. We now copy the memory pointed to by inst1.p to the memory pointed to by inst2.p. You can see exactly how in the copy constructor function of the string class example at the end of this chapter. By copying the memory at the pointers rather than just the pointers, we perform a deep copy and ensure that the pointers don't get corrupted. Memory after the assignment looks like this:



The statement return(*this) returns a reference to the changed contents of inst2 to the assignment. In this way the contents of inst1 are copied to inst2 without the unwanted side-effects referred to above.

Note that, to prevent further pointer corruption, both the parameter and return value of operator= must be references to the operands of the overloaded assignment.

Assignment and initialisation

You saw in the last section that shallow copy or memberwise assignment causes memory corruption. So does initialisation when it's done in either of the two ways:

```
ptrclass inst2(inst1);
ptrclass inst2 = inst1;
```

In each case of initialisation such as this, the compiler generates a default copy mechanism (a *default copy constructor*) that does a blind member-by-member initialisation known as *memberwise initialisation*. This is similar to the memberwise assignment seen in the last section, and it messes up memory in the same ways. You can resolve these problems by providing tailored copy constructors that perform *smart assignment* of pointers which are class members.

I've shown two cases of class instance initialisation. There are two others:

- When a function receives a class instance as an argument.
- When a function returns a class instance.

For all four cases, you need a copy constructor to prevent corruption of memory.

Initialising objects with copy constructors

A copy constructor is one that is called to initialise the class instance of which it is a member to the value of another class instance.

In the case of the class X, the prototype of the copy constructor looks like this:

```
X::X(const X&);
```

If no copy constructor is defined for a class, then initialising operations cause the default copy constructor to be called quietly. The default copy constructor isn't very refined, performing as it does simple memberwise initialisation.

To see a case where we need a copy constructor, let's look at the class coord:

```
class coord
{
private:
    int *x_coord, *y_coord;
public:
    coord(int x, int y)
    {
        cout << "Constructing..." << endl;
        x_coord  = new int;
        *x_coord = x;
        y_coord  = new int;
        *y_coord = y;
    }
    void print()
    {
        cout << *x_coord << " " << *y_coord << endl;
        }
        ~coord()
        {
            cout << "Destructing..." << endl;
            delete x_coord;
            delete y_coord;
        }
    };
```

Defining an instance of coord:

```
    coord point1(5, 10);
```

works fine, with memory to accommodate the arguments 5 and 10 being allocated to the pointers x_coord and y_coord by the constructor function. It's when we try to do either of the (equivalent) initialising operations

```
    coord point2(point1);
    coord point2 = point1;
```

that we run into the shallow-copy memory-corruption problem which I described in the last section.

In both cases, the default copy constructor initialises the pointer values in point2 with those stored in point1. The destructor, which is called twice, then attempts to deallocate the same memory twice. The results of doing this are undefined but are always an error and may cause the program to crash.

The problem is resolved using a specially-written copy constructor, which is added to coord as a function member:

```
    coord(const coord& copypoint)
    {
        cout << "Copy constructing....." << endl;
        x_coord = new int;
        *x_coord = *(copypoint.x_coord);
        y_coord = new int;
        *y_coord = *(copypoint.y_coord);
    }
```

The class point2 is initialised by an explicit call to the copy constructor. In the earlier example, the default copy constructor shallow-copied the pointer values x_coord and y_coord, leading to an attempted double memory deallocation. This time the *integer objects pointed to* by x_coord and y_coord are copied to newly-allocated memory in point2. When the destructor is eventually called twice, it each time deallocates different memory.

With the copy constructor included in the coord class, initialisation of class instances in any of the four ways described at the start of this section will use the copy constructor and not the default copy constructor. The resulting initialisation is error-free.

Example: a C-string class

The C-style character string — strictly, the null-terminated character array, or C-string — is one of the data objects most commonly used in C++ programming. A large number of string operations are also defined, including those provided in the Standard C Library. As shown in Chapter 2, the ISO C++ Standard Library additionally defines a general purpose string class.

Because it illustrates well so many aspects of class implementation and class services in C++, this section presents a cstr class example, in no way intended as an alternative to the standard string class.

First, the cstr class is declared as part of the header file cstr.h:

```
//   cstr.h — defines C-string class
class cstr
{
private:
     char *sptr;
     int  slen;
     int  ssize;
public:
     cstr();
     cstr(int);
     cstr(const char *);
     cstr(const cstr &);
     void set_str(const char *);
     char *access() { return(sptr); }
     // binary operator-overload function for
     // C-string concatenation
     void operator+=(cstr&);
     // overloaded assignment operator, copies C-strings
     cstr& operator=(cstr&);
     ~cstr() { delete sptr; }
};
extern const int MAX;
```

The class cstr defines a character pointer sptr as a private data member, along with length and array-size information. The four constructor functions in different ways allocate space for this pointer and initialise the resulting character array as a C-string. The fourth constructor in the list is the copy constructor for the cstr class. The destructor deallocates memory reserved for cstr instances. The two overloaded operator functions implement C-string concatenation and assignment.

The class defines an access function — called access — to retrieve the value of sptr and the function set_str to set the text value of a cstr instance. The code implementing the four constructors and the other member functions is in the program file cstrfunc.cpp.

```
//   cstrfunc.cpp — defines cstr class functions
#include <iostream>
using namespace std;
#include <cstring>      // Standard C Library string class
#include "cstr.h"   // Our C-string class
// cstr constructors
cstr::cstr()
{
     sptr = new char[MAX];
     ssize = MAX;
     *sptr = '\0';
     slen = 0;
}

cstr::cstr(int size)
{
     sptr = new char[size];
     ssize = size;
     *sptr = '\0';
     slen = 0;
}

cstr::cstr(const char *s_in)
{
     slen = ssize = strlen(s_in) + 1;
     sptr = new char[slen];
     strcpy(sptr, s_in);
}

// copy constructor
cstr::cstr(const cstr& ob_in)
{
     slen = ssize = strlen(ob_in.sptr) + 1;
     sptr = new char[slen];
     strcpy(sptr, ob_in.sptr);
}

void cstr::set_str(const char *s_in)
{
     delete sptr;
     slen = ssize = strlen(s_in) + 1;
     sptr = new char[slen];
     strcpy(sptr, s_in);
}
```

```
void cstr::operator+=(cstr& s2)
{
    char *ap;
    slen += (s2.slen + 1);
    ap = new char[slen];
    strcpy(ap, sptr);
    strcat(ap, s2.sptr);
    delete sptr;
    ssize = slen;
    sptr = new char[slen];
    strcpy(sptr, ap);
}
//  'operator=' — assigns cstrs

cstr& cstr::operator=(cstr& s2)
{
    //  watch for the case of assignment of the same class!
    //  (e.g. s1 = s1 would mean losing the cstr)
    if (this == &s2)
        return(*this);
    //  deallocate cstr space in class object (this) being copied to,
    //  then reallocate enough space for the object being copied
    delete sptr;
    sptr = new char[s2.slen];

    //  copy the cstr and its length
    slen = ssize = s2.slen;
    strcpy(sptr, s2.sptr);
    //  return this class object to the assignment
    return(*this);
}
```

The first constructor allocates to the pointer sptr a character array of fixed length MAX. The second allocates an array of length specified by its parameter. The third allocates an array long enough to accommodate the text of its parameter. All three constructors null-terminate the array and set its length counter.

The fourth constructor is the copy constructor. This first finds the length of the text in the incoming cstr instance. It then allocates enough memory to the pointer in the instance being initialised to accommodate that text. Finally, the text (not the pointers!) is copied.

The overloaded functions **operator+=** and **operator=** similarly copy text contents of C-string instances when such instances are assigned (not initialised) in the main function. The main function calls all the functions, as well as (quietly) the destructor to deallocate memory assigned by the constructors to sptr.

The code that calls the member functions of cstr is in the main function in the program file cstr.cpp

```
//   program file 'cstr.cpp'
#include <iostream>
using namespace std;

#include "cstr.h"

const int MAX = 256;

int main()
{
     cstr s1;
     cstr s2(MAX);
     cstr s3("and into Mary's bread and jam ");
     cstr s4("his sooty foot he put");

     s1.set_str("Mary had a little lamb ");
     s2.set_str("whose feet were black as soot ");

     s1 += s2;               // overloaded '+='
     s1 += s3;
     s1 += s4;

     cstr s5;

     s5 = s1;                // overloaded assignment

     cout << "s5: " << s5.access() << endl;

     cstr s6(s5);              // copy constructor
     cout << "s6: " << s6.access() << endl;

     cstr s7 = s6;             // copy constructor
     cout << "s7: " << s7.access() << endl;
}
```

In essence, the program initialises the cstr instances s1, s2, s3 and s4 and, with these, sets up s5, s6 and s7 using the various constructor and overloaded-operator facilities implemented by the class. When the program is run, the nursery rhyme is each time displayed in full by sending to the output stream the contents of the cstr objects s5, s6 and s7.

Exercises

1 Enumerate, with very short examples, the four cases of initialisation which require a copy constructor.

2 Explain how the chained assignment operation

```
s3 = s2 = s1
```

is implemented, where s1, s2 and s3 are objects of the cstr class.

3 Write a program that overloads the stream insertion operator << such that the operand to its right can be a class instance.

10 Inheritance

Introduction

Class inheritance, with virtual functions, is what C++ is all about. Everything you've learnt up to now in this book is essentially groundwork that you have to cover to be able to take advantage of the programming power offered by classes containing virtual functions, organised in hierarchies. In this chapter, you'll learn how to:

- Derive classes from existing base classes.
- Control access to the data members of derived and base classes.
- Use constructors and destructors to initialise and destroy instances of derived and base classes.
- Derive classes from multiple base classes.
- Use the C++ implementation of polymorphism: the hierarchy of base and derived classes combined with virtual functions.

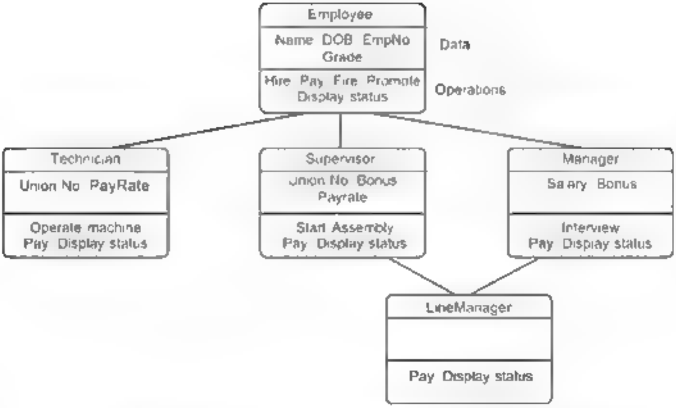
Classes often have much in common with other classes. Where classes are similar, it's better to define them in terms of characteristics they have in common, instead of duplicating them. With class derivation, C++ allows classes to reuse declarations made in other classes. Derived classes inherit the declarations made in existing base classes.

Let's look at an intuitive example of this process, with the **employee** class example. All employees share certain characteristics: they have a name, date of birth, employee number and grade; all employees are also hired, paid and (maybe) fired.

For specific employee types, we need other data and behaviour. A Manager probably has a salary and bonus, rather than the hourly pay of an ordinary employee. A Supervisor may have a union number. A Line Manager may share some of the characteristics of both the Manager and Supervisor. For example, they may both be entitled to use secretarial services. A Director, on the other hand, may have exclusive access to a Personal Assistant. If our company is well off (maybe more likely if it's not!) the Vice President's perks could include a company-paid yacht in the Caribbean or Mediterranean.

In the employee example, it makes sense to define a generic class called **employee**, holding basic information on the characteristics and behaviour of all employees. Class definitions for **supervisor**, **manager** and the others may then be derived from the **employee** class. The model is shown as a diagram on the next page.

If you use this sort of model with C++, you can get impressive code reuse and serious savings in software development cost compared with more traditional languages such as C. In the diagram, the Technician, Supervisor and Manager classes are derived with single inheritance from Employee. LineManager multiply inherits the characteristics of Supervisor and Manager. You can design and build these class hierarchies as deep as you like.



You derive the Supervisor and Manager classes from Employee because they have a lot in common. There are, however, differences of detail in the ways in which similar operations are carried out. For example, all the employees are paid but on different terms and schedules. Displaying the status and qualifications of a director will differ in detail from the equivalent operation for a janitor.

To deal elegantly with implementing these differences, C++ implements polymorphism: the ability to define many different operations that use the same name and present the same interface to the programmer.

C++ implements polymorphism using virtual functions declared in a base class and inherited by one or more derived classes. You can use the same function call to carry out a similar (but different) operation for any of the classes in the hierarchy. You have to define different instances of the function for each operation. When the program runs, the runtime system selects the appropriate instance depending on the class instance used in the function call.

With virtual functions, you get a further level of abstraction: the detailed implementations of the different virtual function instances are hidden and *you don't have to know* what sort of instance (Employee, Manager and so on) you're dealing with in order to, say, pay that employee. With virtual functions, type-checking is reduced and with it the incidence of programmer error.

Class inheritance

Here's a first look at the employee class hierarchy We have declarations of the employee base class and of three derived classes, technician supervisor and manager Don't worry about their full contents yet.

```
class employee
{
    // no private members, but could be
protected
    // members hidden from rest of world
public
    int grade
    // public class members
}
class technician   public employee
{
private
    // class members specific to 'technician'
public
    int unionNo.
    // public member functions can access
    // private members of this class as well
    // as protected members of 'employee'
}
class supervisor   public employee
{
private
    //
public.
    //
}.
class manager   public employee
{
private
    //
public
    //
}.
```

After the base employee class is declared. the declaration

```
class supervisor   public employee
{
//
}
```

announces a new type supervisor which inherits all non private characteristics of employee and between the curly braces adds zero or more declarations of its own You must specify the class keyword, as well as the names of the two class types, separated by a colon

The access specifier **public** is optional but usually necessary. When a class is derived from one or more other classes, and when the access specifier **public** is used in the derived class declaration, public members of the base class become public members of the derived class. If **public** is not specified in this way, the members of the derived class are by default private.

Each of **technician**, **supervisor** and **manager** is declared as a derived class of the base class **employee**. Using an instance of any of the derived class types, you can access all non-private members of the inherited **employee** object as if those members were also members of the derived classes. Here's the code:

```
// define 'technician' and 'employee' class objects

employee    e1;
technician  t1;

// illustrate basic access rules, assuming
// 'public' access specifier in derived-class declarations

e1.grade = 1;           // OK, 'grade' is 'employee'
// member

t1.grade = 1;           // OK, 'grade' is 'technician'
// member derived from 'employee'

t1.unionNo = 53;        // OK, 'unionNo' is 'technician'
// member not derived from 'employee'

e1.unionNo = 7;         // Error, 'unionNo' is not in
// scope for 'employee' object
```

This shows that a derived class inherits all non-private members of a base class and that those members are in scope for the derived class. The converse is not true: new members declared in a derived class are not in scope for the base class.

Example: A simple employee class hierarchy

Here's a full-program example, based on the **employee** model, that illustrates single class inheritance and the C++ syntax used to access the members of the classes in a hierarchy.

The program is organised in three program files. **employee.h** contains the class declarations. The program file **empfunc.cpp** defines the member functions of the class hierarchy and **emp.cpp** the small amount of code needed to define class objects and use their members.

```
 //  employee.h
enum qualification {NONE, CERT, DIPLOMA, DEGREE, POSTGRAD};

class employee
{
protected:
    char *name;
    char *dateOfBirth;
    int individualEmployeeNo;
    static int employeeNo;
    int grade;
    qualification  employeeQual;
    float accumPay;
public:
    // constructor
    employee();

    // destructor
    ~employee();

    void pay();
    void promote(int);      // scale increment
    void displayStatus();
}

class technician : public employee
{
private:
    float hourlyRate;
    int   unionNo;
public:
    // constructor
    technician();
    // destructor
    ~technician();
    void pay();
    void displayStatus();
}

class supervisor : public employee
{
private:
    float monthlyPay;
public:
    // constructor
    supervisor();
```

```
     // destructor
     ~supervisor();
     void pay();
     void displayStatus();
};

class manager : public employee
{
private:
     float monthlyPay;
     float bonus;
public:
     // constructor
     manager();
     // destructor
     ~manager();
     void pay();
     void displayStatus();
};
```

The classes technician, supervisor and manager are derived from the base class employee. All non-private members of employee are inherited by and are common to the derived classes.

All the classes have a constructor and a destructor. The constructors do not yet take parameters. Each class defines its own pay and displayStatus functions. The existence of multiple definitions of these functions among the classes does not cause ambiguity. Any call to, say, the pay function for a given class must, in client code, be qualified with a class instance:

```
// illustrate 'pay' function call
supervisor s1;


s1.pay();   // not ambiguous
```

You can call the function pay without the "s1." prefix from within a member function of technician. In that case, the pay function that is a member of technician is called.

The base class employee, uniquely, contains a declaration for the function promote. The employee instance of this function is called no matter which object type — employee, technician, supervisor or manager — is used to qualify the promote call.

The program file emplfunc.cpp contains the code that implements the member functions of the four classes.

```
    cout << "Hourly employee " << name << " is hired" << endl;
}
technician::~technician()
{
    cout << "Hourly employee " << name << " is fired!" << endl;
}

void technician::pay()
{
    float paycheck;
    paycheck = hourlyRate * 40;
    accumPay += paycheck;
    cout << "Hourly employee " << individualEmployeeNo
        << " paid " << paycheck << endl;
}

void technician::displayStatus()
{
    cout << "Hourly employee " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}

//     define 'supervisor' member functions
supervisor::supervisor()
{
    monthlyPay = 1700.00;
    cout << "Supervisor " << name << " is hired" << endl;
}
supervisor::~supervisor()
{
    cout << "Supervisor " << name << " is fired!" << endl;
}

void supervisor::pay()
{
    accumPay += monthlyPay;
    cout << "Supervisor " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}

void supervisor::displayStatus()
{
    cout << "Supervisor " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}
```

```
//     define 'manager' member functions
manager::manager()
{
    monthlyPay = 2100.00;
    bonus      = 210.0;
    cout << "Manager " << name << " is hired" << endl;
}
manager::~manager()
{
    cout << "Manager " << name << " is fired!" << endl;
}

void manager::pay()
{
    accumPay += monthlyPay;
    cout << "Manager " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}

void manager::displayStatus()
{
    cout << "Manager " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}
```

None of the constructor functions takes any parameters, so the employee constructor must prompt the user for input of employee names. In the typical case, no instances of the base class, employee, will be created. Two of its member functions, pay and displayStatus, therefore have no purpose and are empty.

Here's the main function.

```
//   emp.cpp
#include <iostream>
using namespace std;
#include "employee.h"

int main()
{
    technician t1;
    supervisor s1;
    manager    m1;
    t1.pay();
    t1.displayStatus();
```

```
        s1.pay();
        s1.displayStatus();
        m1.pay();
        m1.displayStatus();
    }
```

Three class objects are defined, one each for technician, supervisor and manager. In each case, an underlying employee object is implicitly defined also. The program produces the following output. Text in bold type is what you enter:

```
Enter new employee name john
Hourly employee john is hired
Enter new employee name chris
Supervisor chris is hired
Enter new employee name marilyn
Manager marilyn is hired
Hourly employee 1000 paid 216
Hourly employee 1000 is of grade 1
    and has been paid 216 so far this year
Supervisor 1001 paid 1700
Supervisor 1001 is of grade 1
    and has been paid 1700 so far this year
Manager 1002 paid 2100
Manager 1002 is of grade 1
    and has been paid 2100 so far this year
Manager marilyn is fired!
Supervisor chris is fired!
Hourly employee john is fired!
```

Access control

I've already explained the effect of the access specifier keywords private and public Now we also look at the protected keyword as well as the levels of access to members of derived classes that are allowed by various combinations of private. protected and public

Base class access

Base class access for a derived class is defined by use of any of the access-specifiers private protected or public

In public derivation

```
class manager   public employee
```

manager inherits protected and public members of employee and retains those access levels

In protected derivation:

```
class manager   protected employee
```

manager inherits protected and public members of employee but forces all the inherited public members to be protected: you can't access them from client code using an employee object

In private derivation:

```
class manager   private employee
```

all non-private members of employee are inherited by manager but are now private members of manager regardless of whether they are specified with protected or public access in employee

Public derivation is the default for structures and unions class derivation defaults to private Here's an example that illustrates many of the possibilities of base class access.

```
class a
{
protected
    int x,
public
    int y;
    int z.
}

class b   private a  // members of a
                     // private in b
{
protected
    a :x;        // x converted to protected
```

Constructors and destructors

This section considers the order in which constructor and destructor members of a class hierarchy are called and the means by which arguments are passed to constructors in the hierarchy. In a class hierarchy formed of a base class and zero or more derived classes, constructor functions are executed starting with the base class in order of class derivation. Destructor functions are called in reverse order of derivation.

Constructor and destructor functions are never inherited. Therefore, in a class hierarchy, the constructor of a derived class does not take on any of the characteristics of the constructor (if any) declared in its base class.

If a base class constructor takes parameters, you can do the initialisation using the syntax shown in Chapter 9. Here's the **employee** base class reworked to declare constructor and destructor functions taking parameters.

```
 class employee

    {
    protected:
        char *name;
        char *dateOfBirth;
        int individualEmployeeNo;
        static int employeeNo;
        int grade;
        qualification employeeQual;
        float accumPay;
    public:
        // constructor: name and grade
        employee(char *, int);

        // constructor: name, birthdate, grade, qualification
        employee(char *, char *, int, qualification);

        // destructor
        ~employee();

        void pay();
        void promote(int);       // scale increment
        void displayStatus();
    }
```

You initialise class instances of type **employee** with definitions like this:

```
    employee e1("Karen", 4);
    employee e2("John", "580525", 4, DEGREE);
```

The first definition creates a class object **e1** of type **employee** and calls the matching constructor function (the one declaring two parameters in its argument list) to initialise the object with the arguments "Karen" and 4.

In a class hierarchy, what you usually want is to initialise a derived class instance using a constructor of that derived class. When you create a derived class instance, you also (quietly) make a base class instance. We need a mechanism to call the derived constructor with arguments and then to transmit some, all or none of those arguments to the base class constructor so that the base member variables may be initialised.

Let's look at creation of a derived-class instance of type technician. The constructors of both the employee and technician classes take parameters. The technician class declaration is this:

```
class technician : public employee
{
private:
    float hourlyRate;
    int   unionNo;
public:
    // name, grade, rate, union ID
    technician(char *, int, float, int);

    // name, birthdate, grade, qualification, rate, union ID
    technician(char *, char *, int, qualification, float, int);

    // destructor
    ~technician();

    void pay();
    void displayStatus();
};
```

You write the header of the second constructor function of the technician class like this:

```
technician::technician(char  *nameIn,
                        char *birthIn,
                        int  gradeIn,
                        qualification qualIn,
                        float rateIn,
                        int  unionNoIn)
            : employee(nameIn, birthIn, gradeIn, qualIn)
```

Four of the six parameters received by the technician constructor arguments are passed on to the matching employee constructor. The technician constructor takes its own parameters, rateIn and unionNoIn, and assigns them to the member variables hourlyRate and unionNo of its class.

Example: Class hierarchy with constructors taking parameters

The full employee class hierarchy, shown with constructors and destructors taking parameters, follows.

```
//   employee.h
enum qualification {NONE, CERT, DIPLOMA, DEGREE, POSTGRAD};

class employee
{
protected:
     char *name;
     char *dateOfBirth;
     int individualEmployeeNo;
     static int employeeNo;
     int grade;
     qualification  employeeQual;
     float accumPay;
public:
     // constructor: name and grade
     employee(char *, int);
     // constructor: name, birthdate, grade, qualification
     employee(char *, char *, int, qualification);
     // destructor
     ~employee();
     void pay();
     void promote(int); // scale increment
     void displayStatus();
}

class technician : public employee
{
private:
     float hourlyRate;
     int   unionNo;
public:
     // name, grade, rate, union ID
     technician(char *, int, float, int);
     // name, birthdate, grade, qualification, rate, union ID
     technician(char *, char *, int, qualification, float, int);
     // destructor
     ~technician();
     void pay();
     void displayStatus();
}

class supervisor : public employee
```

```
{
private:
    float monthlyPay;
public:
    // name, grade, rate
    supervisor(char *, int, float);
    // name, birthdate, grade, qualification, rate
    supervisor(char *, char *, int, qualification, float);
    // destructor
    ~supervisor();
    void pay();
    void displayStatus();
};

class manager : public employee
{
private:
    float monthlyPay;
    float bonus;
public:
    // name, grade, rate, bonus
    manager(char *, int, float, float);
    // name, birthdate, grade, qualification, rate, bonus
    manager(char *, char *, int, qualification, float, float);
    // destructor
    ~manager();
    void pay();
    void displayStatus();
};
```

We implement the member functions of all four classes in the program file empfunc.cpp

```
//  empfunc.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "employee.h"

//  define and initialise static member
int employee::employeeNo = 1000;

//  define 'employee' member functions first
employee::employee(char *nameIn, int gradeIn)
{
    name = new char[strlen(nameIn) + 1];
    strcpy(name, nameIn);
    dateOfBirth = NULL;
    individualEmployeeNo = employeeNo++;
    grade = gradeIn;
    employeeQual = NONE;
    accumPay = 0.0;
}

employee::employee(char *nameIn,
                   char *birthIn,
                   int gradeIn,
                   qualification qualIn)
{
    name = new char[strlen(nameIn) + 1];
    strcpy(name, nameIn);
    dateOfBirth = new char[strlen(birthIn) + 1];
    strcpy(dateOfBirth, birthIn);
    grade = gradeIn;
    employeeQual = qualIn;
    individualEmployeeNo = employeeNo++;
    accumPay = 0.0;
}

employee::~employee()
{
    delete name;
    delete dateOfBirth;
}

void employee::pay()
{
}

void employee::promote(int increment)
{
    grade += increment;
}

void employee::displayStatus()
{
}

//  define 'technician' member functions
technician::technician(char *nameIn,
                       int  gradeIn,
                       float rateIn,
                       int  unionNoIn)
```

```
            employee(nameIn, gradeIn)
{
    hourlyRate = rateIn;
    unionNo    = unionNoIn;
    cout << "Hourly employee " << name << " is hired" << endl;
}

technician::technician(char  *nameIn,
                        char *birthIn,
                        int  gradeIn,
                        qualification qualIn,
                        float rateIn,
                        int  unionNoIn)
            : employee(nameIn, birthIn, gradeIn, qualIn)
{
    hourlyRate = rateIn;
    unionNo    = unionNoIn;
    cout << "Hourly employee " << name << " is hired" << endl;
}

technician::~technician()
{
    cout << "Hourly employee " << name << " is fired!" << endl;
}

void technician::pay()
{
    float paycheck;
    paycheck = hourlyRate * 40;
    accumPay += paycheck;
    cout << "Hourly employee " << individualEmployeeNo
        << " paid " << paycheck << endl;
}

void technician::displayStatus()
{
    cout << "Hourly employee " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}

//  define 'supervisor' member functions
supervisor::supervisor(char  *nameIn,
                        int  gradeIn,
                        float rateIn)
            : employee(nameIn, gradeIn)
```

```
{
    monthlyPay = rateIn;
    cout << "Supervisor " << name << " is hired" << endl;
}

supervisor::supervisor(char *nameIn,
                         char *birthIn,
                         int  gradeIn,
                         qualification qualIn,
                         float rateIn)
             : employee(nameIn, birthIn, gradeIn, qualIn)
{
    monthlyPay = rateIn;
    cout << "Supervisor " << name << " is hired" << endl;
}

supervisor::~supervisor()
{
    cout << "Supervisor " << name << " is fired!" << endl;
}

void supervisor::pay()
{
    accumPay += monthlyPay;
    cout << "Supervisor " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}

void supervisor::displayStatus()
{
    cout << "Supervisor " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}

//  define 'manager' member functions
manager::manager(char *nameIn,
                   int  gradeIn,
                   float rateIn,
                   float bonusIn)
          : employee(nameIn, gradeIn)
{
    monthlyPay = rateIn;
    bonus      = bonusIn;
    cout << "Manager " << name << " is hired" << endl;
}
```

```
manager::manager(char *nameIn,
                 char *birthIn,
                 int  gradeIn,
                 qualification qualIn,
                 float rateIn,
                 float bonusIn)
          employee(nameIn, birthIn, gradeIn, qualIn)
{
    monthlyPay = rateIn;
    bonus      = bonusIn;
    cout << "Manager " << name << " is hired" << endl;
}

manager::~manager()
{
    cout << "Manager " << name << " is fired!" << endl;
}

void manager::pay()
{
    accumPay += monthlyPay;
    cout << "Manager " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}

void manager::displayStatus()
{
    cout << "Manager " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}
```

The main function drives the classes and their member functions.

```
//  emp.cpp
#include <iostream>
using namespace std;
#include "employee.h"

int main()
{
    technician t1("Mary", 1, 10.40, 1234);
    technician t2("Jane", "651029", 2, CERT, 10.40, 1235);
    supervisor s1("Karen", 4, 1350.00);
    supervisor s2("John", "580525", 4, DEGREE, 1700.00);
    manager m1("Susan", 6, 1350.00, 150.00);
```

```
        manager m2
        ("Martin", "580925", 5, POSTGRAD, 1700.00, 200.00);
        t1.pay();
        t1.displayStatus();
        t2.pay();
        t2.displayStatus();
        s1.pay();
        s1.displayStatus();
        s2.pay();
        s2.displayStatus();
        m1.pay();
        m1.displayStatus();
        m2.pay();
        m2.displayStatus();
    }
```

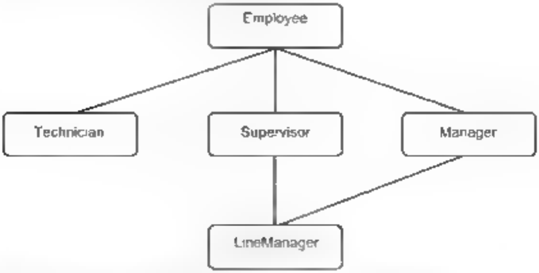
When you run the program, the output is this.

```
Hourly employee Mary is hired
Hourly employee Jane is hired
Supervisor Karen is hired
Supervisor John is hired
Manager Susan is hired
Manager Martin is hired
Hourly employee 1000 paid 216
Hourly employee 1000 is of grade 1
    and has been paid 216 so far this year
Hourly employee 1001 paid 216
Hourly employee 1001 is of grade 2
    and has been paid 216 so far this year
Supervisor 1002 paid 1350
Supervisor 1002 is of grade 4
   and has been paid 1350 so far this year
Supervisor 1003 paid 1700
Supervisor 1003 is of grade 4
   and has been paid 1700 so far this year
Manager 1004 paid 1350
Manager 1004 is of grade 6
   and has been paid 1350 so far this year
Manager 1005 paid 1700
Manager 1005 is of grade 5
   and has been paid 1700 so far this year
Manager Martin is fired!
Manager Susan is fired!
Supervisor John is fired!
Supervisor Karen is fired!
Hourly employee Jane is fired!
Hourly employee Mary is fired!
```

Multiple inheritance

Up to now, we have considered inheritance by derived classes only of single base classes. A derived class can inherit the characteristics of more than one base class. This facility of C++ reflects and accommodates real-world objects that you may want to simulate.

This book (being, recall, a *Made Simple*), confines itself to a simple general presentation of multiple inheritance. We could apply the technique to the employee class by, for example, actually implementing the **lineManager** class. This is derived from both **supervisor** and **manager**, which in turn have the single base class **employee**. However, multiple inheritance raises a number of complexities and difficult issues which are really beyond the scope of this book. So, if you want to know how to propagate constructor parameters within a multiply-inherited hierarchy, or how to resolve the ambiguity (two instantiations of **employee** for one definition of **lineManager**) in this hierarchy



then have a look at the *C++ Users Handbook*, or the *C++ Programming Language* (3rd edn) by Stroustrup. If you don't want to know this stuff, feel happy about it. Most C++ programs are written with little or no multiple inheritance. There's even a strong body of opinion which holds that multiple inheritance is a Bad Thing and is never necessary. And, in fact, it's very difficult to contrive a class hierarchy where multiple inheritance is *unavoidable*. So, with all that rationalisation done, let's look at the essence of the thing.

Suppose a class **d** is to be declared that inherits the classes **a**, **b** and **c**. Classes **a** and **c** are to be inherited by **d** with public access, and **b** with private access. Here is the syntax for declaration (with multiple inheritance) of class **d**:

```
class d : public a, private b, public c
{
// 'class d' declarations
};
```

The constructor functions of a singly-inherited hierarchy of classes are executed in order of class derivation. The same is true for hierarchies containing classes derived from multiple bases.

If the base classes have constructor functions, the constructors are executed, left to right, in the same order as that in which the base classes are specified. Destructors are invoked in the reverse order. This is a generalisation of the execution-order rules given in the last section, as a simple example shows:

```
#include <iostream>
using namespace std;

class base
{
public:
     base() { cout << "Constructing 'base'" << endl; }
     ~base() { cout << "Destructing 'base'" << endl; }
};

class a : public base
{
public:
     a() { cout << "Constructing 'a'" << endl; }
     ~a() { cout << "Destructing 'a'" << endl; }
};

class b
{
public:
     b() { cout << "Constructing 'b'" << endl; }
     ~b() { cout << "Destructing 'b'" << endl; }
};

class c
{
public:
     c() { cout << "Constructing 'c'" << endl; }
     ~c() { cout << "Destructing 'c'" << endl; }
};

class d : public a, public b, public c
{
public:
     d() { cout << "Constructing 'd'" << endl; }
     ~d() { cout << "Destructing 'd'" << endl; }
};
```

```
int main()
{
    d d1;   // define instance of 'd'
}
```

Here we have a base class **base**, from which **a** is derived. Classes **b** and **c** are separately declared and **a**, **b** and **c** in turn are base classes of **d**. When an instance, **d1**, of class **d** is defined in the **main** function, the constructors are invoked in the order of derivation and the destructors are executed in reverse order. The order can be traced from the program's output.

```
Constructing 'base'
Constructing 'a'
Constructing 'b'
Constructing 'c'
Constructing 'd'
Destructing 'd'
Destructing 'c'
Destructing 'b'
Destructing 'a'
Destructing 'base'
```

Any or all of the constructor functions in a class hierarchy containing classes derived from multiple bases may require arguments. The order of execution of the constructors and destructors can be described as left-to-right, top-to-bottom. If you define an instance of **lineManager**, the order of constructor calls will be this:

```
employee
supervisor
employee
manager
lineManager
```

Constructor parameters are transmitted 'up the hierarchy' in a manner which is a logical extension to that you've already seen in Section 10.4.

However, if you call the virtual function with a pointer or reference to a class object, the instance of the function called is the instance defined by that class object.

```
ep = &m1;
ep->pay();    // call 'manager' copy of virtual 'pay'
```

All redefinitions in derived classes of a virtual function must have argument lists identical to those of the base declaration of the virtual function.

Hierarchy with virtual functions

Here's a virtual version of a simplified employee hierarchy with a full program implementing it. In the base class employee the pay function is prefixed with the keyword virtual:

```
//  employee.h
enum qualification
    {NONE, CERT, DIPLOMA, DEGREE, POSTGRAD};
class employee
{
protected:
    char *name;
    char *dateOfBirth;
    int individualEmployeeNo;
    static int employeeNo;
    int grade;
    qualification employeeQual;
    float accumPay;
public:
    // constructor
    employee();
    virtual void pay(); // virtual function!
};

class technician : public employee
{
private:
    float hourlyRate;
    int  unionNo;
public:
    // constructor
    technician();

    void promote(int);      // scale increment
    void pay();
};

class supervisor : public employee
{
private:
    float monthlyPay;
public:
    // constructor
    supervisor();
    void pay();
};
```

```
class manager : public employee
{
private:
    float monthlyPay;
    float bonus;
public:
    // constructor
    manager();

    void pay();
}
```

The member functions of the classes are implemented in the program file empfunc.cpp

```
//  empfunc.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "employee.h"

//      define and initialise static member
int employee::employeeNo = 1000;
//      define 'employee' member functions first
employee::employee()
{
    char nameIn[50];
    strcpy(nameIn, "Base Employee");
    name = new char[strlen(nameIn) + 1];
    strcpy(name, nameIn);
    dateOfBirth = NULL;
    individualEmployeeNo = employeeNo++;
    grade = 1;
    employeeQual = NONE;
    accumPay = 0.0;
}

void employee::pay()
{
    cout << "Base-class employee paid!" << endl;
}

//      define 'technician' member functions
technician::technician()
{
    strcpy(name, "Technician");
    hourlyRate = 10.4;
    unionNo    = 0;
}

void technician::promote(int increment)
{
    grade += increment;
}

void technician::pay()
{
    float paycheck;
    paycheck = hourlyRate * 40;
    accumPay += paycheck;
    cout << "Technician paid!" << endl;
}

//    define 'supervisor' member functions
supervisor::supervisor()
{
    strcpy(name, "Supervisor");
    monthlyPay = 1700.00;
}

void supervisor::pay()
{
    accumPay += monthlyPay;
    cout << "Supervisor paid!" << endl
}

//    define 'manager' member functions
manager::manager()
{
    strcpy(name, "Manager");
    monthlyPay = 2100.00;
    bonus      = 210.0;
}

void manager::pay()
{
    accumPay += monthlyPay;
    cout << "Manager paid!" << endl;
}
```

Code in the main function is used to exercise the classes.

```
//   emp.cpp
#include <iostream>
using namespace std;
#include "employee.h"

int main()
{
    employee e1;
    technician t1;
    supervisor s1;
    employee *ep = &e1;
    technician *tp = &t1;
    supervisor *sp = &s1;
    ep->pay(); // call base-class 'pay'
    ep = &t1;
    ep->pay(); // call 'technician' 'pay'
    ep = &s1;
    ep->pay(); // call 'supervisor' 'pay'
}
```

When you run the program, the results are these:

```
Base-class employee paid!
Technician paid!
Supervisor paid!
```

After the first line is output, the base class object pointer **ep** is assigned the address of the **technician** class object **t1**:

```
ep = &t1
```

Because **ep** has been assigned a pointer of type **technician ***, then, when the function call is made

```
ep->pay();
```

the redefinition of the **pay** function contained in the derived class **technician** is selected at runtime and executed. When **ep** is assigned the address of the **supervisor** class object **s1**, the function call **ep->pay()** causes **supervisor::pay()** to be selected at runtime and executed.

A redefinition of a virtual function in a derived class is said to *override*, rather than overload, the base class instance of the function. The difference is important because, unlike in the case of ordinary function overloading, the resolution of virtual function calls is done at runtime. The process is referred to as late, or dynamic, binding.

It's OK for a derived class not to override a virtual function defined in its base class. In such a case, the base class instance of the function is called even if a pointer to the derived class is used in the function call.

Virtual functions are inherited through multiple levels in a derived class hierarchy. If a virtual function is defined only in a base class, that definition is inherited by all the derived classes.

Another difference between virtual functions and overloaded functions is that virtual functions must be class members, while overloaded functions do not have to be.

Abstract classes

As part of the process of designing a class hierarchy, you often have to declare a base class that itself serves no useful purpose. Such a base class is usually the common denominator of more concrete classes derived from it.

The **employee** class is quite a good example of such a class. If you think about it, you've never seen an *employee*. You've seen a *manager*, a *secretary*, a *supervisor* and so on, but never an abstract *employee*. Even the operation of paying the **employee** by calling the function employee::pay() does not mean very much, it being more common to pay real rather than generic employees. This is why **employee**::pay() is left empty in the first examples of this chapter.

The function **employee**::pay() currently displays the message

```
Base-class employee paid!
```

Most likely, in a real program it would do nothing and be defined only to serve as a base virtual function for the pay functions in the derived classes, which actually do some processing. Where there is a dummy virtual function like this, a different declaration can be used and the dummy definition discarded:

```
virtual void pay() = 0;
```

Now there is no instance of **employee**::pay() and the declaration is called a *pure virtual function*. The pure virtual function must be overridden by one or more functions in a derived class. Formally, a class that contains at least one pure virtual function is called an *abstract class*.

Exercises

1 In this derived class hierarchy:

```
class a
{
public:
    int i;
    //
};
class b : public a
{
public:
    char a;
    //
};
class c : public b
{
public:
    double d;
    //
};
```

fill in the constructor functions necessary to initialise i, a and d following the creation of an object of type c:

```
c c_inst(1.732, 'x', 5);
```

2 Given the class hierarchy:

```
class employee
{
protected:
    int grade;
    int employeeNo;
    char name[30];
public:
    void pay();
    void promote();
};

class manager : public employee
{
private:
    double bonus;
    double payRate;
public:
    void payBonus();
};
```

add to each class a constructor function. In the case of employee, the constructor should take three parameters; the manager constructor should take five parameters

Each constructor should assign values to the data members of its class. Show how the arguments used in the definition of an instance of the class manager are distributed between the manager and employee constructors.

11 Advanced facilities

More on function templates

Function templates are introduced in Chapter 3 as part of the general treatment of C++ (and C) functions. The template introduction at that point is simple (it's at the start of a *Made Simple* book after all), concentrating in essence on the min template example. In this section I present material on function templates that you will need to exploit the capabilities of class templates, dealt with from the next section onward.

Function template parameter list

In the function template declaration,

```
template<class num>
num min(num n1, num n2);
```

<class num> is the template's formal parameter list. The keyword class in this context means *type parameter following*. It's customary to use class here: it was introduced by the pre-ISO C++ language as a way of avoiding addition of another reserved keyword to the language. But the dual use tends to be confusing, and ISO C++ introduced the keyword typename to replace class in this context (class is still valid).

The type parameter may be any basic or user-defined type. You must always use either the class or typename keyword in a template parameter list. If there is more than one type parameter, class/typename must be used for each parameter. Each parameter in the list must be unique and must appear at least once in the argument list of the function. These points are illustrated by modifying the min template:

```
// legal template
template<typename num1, typename num2>
num min(num1 n1, num2 n2)
{
    //
}
// illegal, missing typename
template<typename num1, num2>
num min(num1 n1, num2 n2)
// illegal, duplication
template<typename num1, typename num1, typename num2>
num min(num1 n1, num2 n2)
// illegal, num3 not used
template<typename num1, typename num2, typename num3>
num min(num1 n1, num2 n2)
```

The names of the template type parameters don't have to match in the template declaration and definition:

```
//  declaration
template<typename x, typename y, typename z>
num min(x n1, y n2, z n3)

//  definition
template<typename num1, typename num2, typename num3>
num min(num1 n1, num2 n2, num3 n3)
{
    //
}
```

Declaration and definition

You must declare, if not also define, a function template at a point in the code before a template function is instantiated. If you do this, you can define the template later; the min example in Chapter 3 uses this approach. As with any ordinary function, a function template's definition is its declaration if the definition precedes the first function call. The first call to the function following the definition instantiates a template function.

Both the declaration and definition of a function template must be in global scope. A function template cannot be declared as a member of a class.

User-defined argument types

You can use class types, as well as other user defined types, in the parameter list of a function template and in a call to a template function. If you do this, you must overload basic operators used within the template function on class arguments. Here's an example program, ftmpovl1.cpp:

```
#include <iostream>
using namespace std;

class coord
{
private:
    int x_coord;
    int y_coord;
public:
    coord(int x, int y)
    {
        x_coord = x;
        y_coord = y;
    }
    int GetX() { return(x_coord); }
    int GetY() { return(y_coord); }
    int operator<(coord& c2);
};

//  function template declaration. Use 'lesser', the obvious 'min'
//  is used by the C++ system for another purpose
template<typename obj>
obj& lesser(obj& o1, obj& o2);

int main()
{
    coord c1(5,10);
    coord c2(6,11);

    //  compare coord objects in min,
    //  using overloaded < operator
    coord c3 = lesser(c1, c2);
    cout << "minimum coord is: " << c3.GetX() << " " << c3.GetY() << endl;

    double d1 = 3.14159;
    double d2 = 2.71828;

    //  compare double objects in lesser,
    //  using basic < operator
    cout << "minimum double is: " << lesser(d1, d2) << endl;
}
template<typename obj>
obj& lesser(obj& o1, obj& o2)
{
    //  < operator overloaded if function instantiated for typename type,
    //  otherwise built-in < used
    if (o1 < o2)
        return (o1);
    return (o2);
}
//  define overloaded < operator
int coord::operator<(coord& c2)
{
    if (x_coord < c2.x_coord)
        if (y_coord < c2.y_coord)
            return (1);
    return (0);
}
```

Class templates

The class template is actually a generalisation of the function template. With them you can build *collections* of objects of any type *using the same class template*. Where, in conventional C++, you could have a class of floating point numbers or a class of integers, with class templates, you can define a single number class that caters for both types.

You declare a class template by prefixing a class declaration with a template specification. This is the template keyword followed by a pair of angle-brackets containing one or more identifiers that represent parameterised types or constant initialising values.

Using class templates, you can declare and define a class in terms of any type. Such a class is said to be parameterised. If classes generalise objects, then class templates can be said to generalise classes. Let's look at the code implementing our generic number class.

```
// class template declaration
template <typename numtype>
class number


// definition of a class instance
number<int> ni;



// typename template definition
template <typename numtype>
class number
{
private:
    numtype n;
public:
    number()
    {
        n = 0;
    }
    void get_number() { cin >> n; }
    void print_number() { cout << n << endl; }
}
```

In this situation, with conventional C++, you'd usually have to take the 'brute force' approach and declare a class type for every type of number that you need. With the class template shown, you can instantiate that class for a number of any type. Instantiation occurs when the template name is used with its list of parameters. You define an instance of the class for integer numbers like this:

```
number<int> ni;
```

Now the identifier ni is a class object of type number<int> that specifies the characteristics of an integer number. The definition causes the built-in type specifier int to be substituted for the class template parameter numtype and to be used thereafter in the class declaration in place of numtype. This is exactly as if you explicitly made the class declaration.

```
class number
{
private:
     int n;
public:
    number()
    {
        n = 0;
    }
    void get_number() { cin >> n; }
    void print_number() { cout << n << endl; }
}
```

and defined the instance ni in the ordinary way

```
number ni;
```

Number class template

Here's the full number class program.

```
#include <iostream>
using namespace std;
template <typename numtype>
class number
{
private:
    numtype n;
public:
    number()
    {
        n = 0;
    }
    void get_number() { cin >> n; }
    void print_number() { cout << n << endl; }
};
int main()
{
    number<char> nc;
    cout << "Enter a character: ";
    nc.get_number();
```

```
        cout << "Character is: ";
        nc.print_number();
        number<int> ni;
        cout << "Enter an integer: ";
        ni.get_number();
        cout << "Integer is: ";
        ni.print_number();
        number<double> nd;
        cout << "Enter a double: ";
        nd.get_number();
        cout << "Double is: ";
        nd.print_number();
}
```

We make three template class instantiations, one each for char, int and double types. For each instance, we define the private member n in turn as char, int and double. The member function get_number extracts a value from the standard input stream and stores it in n. The first time it's called, cin uses the extractor that has a standard overloading for type char and expects a character to be input. On the second call to get_number, cin expects input of an int and on the third call a double. If you don't input the numbers in this order, the input operation fails. When you run the program, you get this input-output sequence (user input in boldface):

```
Enter a character: r
Character is: r
Enter an integer: 7
Integer is: 7
Enter a double: 2.64575
Double is: 2.64575
```

Class template syntax

The syntax of class templates appears daunting. While it sure isn't easy, all template syntax has an equivalent usage for simple classes. The basic equivalence is:

```
number<int>    ni; //   instance of template class
number  ni;        //   instance of non-template class 'number'
```

For the template declaration template<typename numtype> class number, the class name number is a parameterised type and numtype (when replaced by a type specifier) is its parameter. Therefore:

```
number<double>
```

is a type specifier that you can use to define a double instance of the template class number in any part of the program for which the template is in scope. Within the template definition, you can use the type specifier number as a shorthand for number<numtype>. Outside the template definition, the type specifier must be used in its full form. If you define the function get_number outside rather than within the template, you must use this function declaration and definition:

```
// function declaration in template
void get_number();

// function definition externally
template <typename numtype>
void number<numtype>::get_number()
{
    cin >> n;
}
```

and the definition is, of course, that of a function template, which we saw in the last section. The header syntax is complex but may make sense when we see that the equivalent non-template header is:

```
void number::get_number()
```

You must prefix the definition of the template function get_number with the template specification template<typename numtype> and specify it as being in the scope of the type number<numtype>.

Class templates obey the normal scope and access rules that apply to all other C++ class and data objects. You must define them in file scope (never within a function) and make them unique in a program. Class template definitions must not be nested.

Class template parameter list

In the class template declaration:

```
template<typename numtype>
class number
```

<typename numtype> is the template's formal parameter list. The type parameter may be any C++ basic or user-defined type. You must use the typename (or class) keyword for each type specified in a parameter list. If there is more than one type parameter, typename must be used for each parameter. A class template parameter list can also contain *expression parameters*, usually numeric values. The arguments supplied to these parameters on instantiation of a template class must be constant expressions. The class template parameter list must not be empty and, if there is more than one parameter, they must be individually separated by commas:

```
template <typename T1, int exp1, typename T2>
class sometypename
{
    //
}
```

```
public:
    array(int slots = 1)
    {
        size = slots;
        aptr = new slottype[slots];
    }
    void fill_array();
    void disp_array();
    ~array() { delete [] aptr; }
};

template <typename slottype>
class term_array : public array<slottype>
{
public:
    term_array(int slots) : array<slottype>(slots)
    {
    }
    void terminate();
    void disp_term_array();
};
```

We define the derived term_array class template with the same parameter list as array and with the class type array<slottype> publicly derived.

Within the term_array class template, the constructor header takes a single argument, slots, from the instantiation of term_array and passes that argument along to the constructor of the base class template array. You must give the type of the base class outside the scope of that class as array<slottype>.

Finally, the term_array template declares two member functions that operate on instances of the derived class template term_array. We define the member functions of the array class template as before:

```
template <typename slottype>
void array<slottype>::fill_array()
{
    for (int i = 0; i < size; i++)
    {
        cout << "Enter data: ";
        cin >> aptr[i];
    }
}

template <typename slottype>
void array<slottype>::disp_array()
{
    for (int i = 0; i < size; i++)
        cout << aptr[i] << " ";
        cout << endl;
    }
```

We define the term_array member functions similarly, specifying that they are in the scope term_array<slottype>

```
    template <typename slottype>
    void term_array<slottype>::disp_term_array()
    {
        cout << "Contents of terminated array are: ";
        for (int i = 0; aptr[i] != (slottype)0 ; i++)
            cout << aptr[i] << endl
    }
    template <typename slottype>
    void term_array<slottype>::terminate()
    {
        cout << "Null terminating the array"
                << endl;
        aptr[size] = (slottype)0;
    }
```

The main function defines an instance of the derived class template term_array. It then calls the fill_array member function of the base class array to accept input values and to store those values in the array. This is an operation common to all arrays.

The characteristics of terminated arrays which are additional to those of general arrays are dealt with by the term_array member functions terminate and disp_term_array. The terminate function null-terminates the array and disp_term_array displays it using the insertion operator for the basic type in use.

```
    int main()
    {
        term_array<char> ac(10);

        cout << "Fill a character array" << endl
        ac.fill_array();
        ac.terminate();
        ac.disp_term_array();

        array<double> ad(5);
        cout << "Fill a double array" << endl;
        ad.fill_array();
        cout << "Array contents are: "
        ad.disp_array();
    }
```

```
int main()
{
        int flag = 1;
        try
        {
                throw_test(flag);
        }
        catch(const char * p)
        {
                cout << "Into character catch-handler" << endl;
                cout << p << endl;
        }
        catch(ob& ob_inst)
        {
                cout << "Into object catch-handler" << endl;
                cout << "Member value is " << ob_inst.member << endl;
        }
}

void throw_test(int flag)
{
        nest1(flag);
}

void nest1(int flag)
{
        nest2(flag);
}
void nest2(int flag)
{
        if (flag == 1)
                throw "Panic!!!";
        else
        if (flag == 2)
        {
                ob ob_inst;
                ob_inst.member = 5;

                throw ob_inst;
        }
}
```

In this case, throw_test calls nest1, which in turn calls nest2. All three functions are subject to the try block and the exceptions thrown from nest2 are caught by the catch handlers following that block. The output results of the program are the same as those for the previous example.

Catch-handler selection

The matching catch handlers closest to the thrown exceptions are those invoked as we can see from the following example

```
// except3.cpp
#include <iostream>
using namespace std;
void nest1(int);
void nest2(int);
void throw_test(int);
class ob
{
public:
        int member;
};

int main()
{
        int flag = 1;
        try
        {
                throw_test(flag);
        }
        catch(const char * p)
        {
                cout << "Into 'main' character catch-handler" << endl;
                cout << p << endl;
        }
        catch(ob& ob_inst)
        {
                cout << "Into object catch-handler" << endl;
                cout << "Member value is " << ob_inst.member << endl;
        }
}

void throw_test(int flag)
{
        try
        {
                nest1(flag);
        }
        catch(const char * p)
        {
                cout << "Into 'throw_test' character catch-handler" << endl;
                cout << p << endl;
        }
}
```

```
void nest1(int flag)
{
        nest2(flag);
}

void nest2(int flag)
{
        if (flag == 1)
                throw "Panic!!!";
        else
        if (flag == 2)
        {
                ob ob_inst;
                ob_inst.member = 5;

                throw ob_inst;
        }
}
```

Here both main and throw_test contain try blocks, while the nested function nest2 generates the exceptions. If nest2 throws a character-string exception the matching catch handler in throw_test is invoked. If it throws an exception of type ob the effect is to call the second catch handler in main. Here are the output results of the program

```
Into 'throw_test' character catch-handler
Panic!!!
```

Finally, throw used without an exception specification

```
throw;
```

causes the most recently thrown exception to be re-thrown to the catch handlers following the nearest try block

Run time type identification

One of the most recent major extensions to the C++ language as it was originally conceived is run time type identification, usually referred to as *RTTI*. In essence, you apply the facilities of RTTI to a given class instance to determine its type. The typical usages are:

- Checking that a given pointer is of a type derived from a specified base type
- Identifying the actual type of a pointer

RTTI should be used sparingly and with care. The whole point of the inheritance and virtual function mechanisms described in Chapter 10 is that you *need not* know the type of a derived-class pointer in order to use it to call a virtual member function of that derived class. RTTI runs contrary to polymorphism and is easy to use badly, allowing degeneration into an alternative form of multi-way switch construct:

```
if (typeid(d1) == typeid(supervisor))
    cout << "It's a supervisor" << endl;
else
if (typeid(d1) == typeid(manager))
    cout << "It's a manager" << endl;
else
if (typeid(d1) == typeid(lineManager))
    cout << "It's a line manager" << endl;
```

Using RTTI is OK where, for a particular type of derived class, an exception needs to be made. The problem inherent in this can be stated: "Given a base class pointer previously assigned an unknown value, how can we ascertain that it points to an instance of the base class or one of its derived classes and, further, how can we determine its actual type?" The answer in both cases, with traditional C++, is that we can't. Enter RTTI.

Identifying derived class objects

To illustrate RTTI, we use a modified form of the **employee** class hierarchy introduced in Chapter 10. From the main function, we pass a base class pointer as an argument to a global function. That function must determine whether or not the pointer holds a pointer value of derived-class type. If it does, then in the case of managers, the employee is paid. Striking **supervisors**, on the other hand, are not paid.

```
//      employee.h
enum qualification {NONE, CERT, DIPLOMA, DEGREE, POSTGRAD};
class employee
{
protected:
    char *name;
    char *dateOfBirth;
    int individualEmployeeNo,
    static int employeeNo;
    int grade,
    qualification  employeeQual;
    float accumPay;
public
    // constructor
    employee();

    // destructor
    ~employee();

    virtual void pay();
    void promote(int);      // scale Increment
    void displayStatus();
},
class supervisor   public employee
{
private
    float monthlyPay;
public.
    // constructor
    supervisor();

    // destructor
    ~supervisor():

    void pay();
    void displayStatus();
};
class manager   public employee
{
private
    float monthlyPay;
    float bonus,
public.
    // constructor
    manager(),
    // destructor
    ~manager(),
    void pay(),
    void displayStatus();
};
//      Global function to demonstrate RTTI
void pay_managers_only(employee *);
```

```
{
    monthlyPay = 1700.00;
    cout << "Supervisor " << name << " is hired" << endl;
}
supervisor::~supervisor()
{
    cout << "Supervisor " << name << " is fired!" << endl;
}

void supervisor::pay()
{
    accumPay += monthlyPay;
    cout << "Supervisor " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}
void supervisor::displayStatus()
{
    cout << "Supervisor " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}

//    define 'manager' member functions
manager::manager()
{
    monthlyPay = 2100.00;
    bonus      = 210.0;
    cout << "Manager " << name << " is hired" << endl;
}
manager::~manager()
{
    cout << "Manager " << name << " is fired!" << endl;
}
void manager::pay()
{
    accumPay += monthlyPay;
    cout << "Manager " << individualEmployeeNo
        << " paid " << monthlyPay << endl;
}
void manager::displayStatus()
{
    cout << "Manager " << individualEmployeeNo
        << " is of grade " << grade << " and has been paid "
        << accumPay << " so far this year" << endl;
}
```

```
Manager 1001 paid 2100
Manager peter is fired!
Supervisor susan is fired!
```

As well as comparing with the base class type, you can determine the precise type of an object. The typeid() operator yields the actual type, not just the information that a given object is or is not of a type included in a class hierarchy. A simple example of typeid() in use follows in the modified file emp.cpp.

```
//      emp.cpp
#include <iostream>
using namespace std;

#include <typeinfo>
#include "employee.h"

int main()
{
    supervisor s1;
    manager    m1;
    employee   *ep = &s1;

    pay_managers_only(ep);

    ep = &m1;

    pay_managers_only(ep);
}

void pay_managers_only(employee *base)
{
    if (typeid(*base) == typeid(manager))
        base->pay();
}
```

The main function is unchanged. The function pay_managers_only now does an explicit comparison of types in deciding whether or not to pay the employee. typeid() returns a reference to library class type_info. This class is declared in the standard header file typeinfo, which must be included for the type_info data to be accessible in client code.

The internal specification of typeinfo is implementation-dependent but it minimally provides overloaded assignment and == operators, as well as a function to return a character pointer to the name of the type found in the call to typeid().

Exercises

1. Write down the declaration of a class template array<arraytype>. Make instantiations of template classes for the int, char and double types.

   Show how a member function is declared within the class template and defined externally.

2. Declare and define a function template called max that you can use to find the maximum of two objects of *any* type. For non numeric objects, what might max mean? Show how overloading can be used to define this.

12 The Standard Library

The ISO C++ Standard Library

The C++ Standard Library is a comprehensive set of tools for use by C++ programmers, itself implemented with the C++ language and, in particular, with templates. The Library was approved for inclusion in the (then) ANSI C++ Draft Standard in July 1994 and subsequently became part of the ISO C++ Standard.

Generations of C programmers became familiar with the internals of implementing such data structures as linked lists, queues, stacks and trees. The C Standard Library (see summary in Chapter 14) provides a large number of C functions that hide some of the smallest details of C programming from the programmer – you can use the **strlen** function to find the length of a C string, not count the characters one-by-one yourself – but the level of abstraction provided is not very high. The C++ Library goes to a much higher level. The programmer can create, manipulate and destroy strings without having to know how such strings are internally implemented or (crucially) how their memory is allocated or deallocated. The programmer can also create instances of standard data structures, such as lists and stacks, without having to know about how this is internally done.

An analogy may be useful. Think of the notion of a collection, in a general sense. A bus queue is a collection (of people, maybe guide-dogs). A shopping-basket full of grocery items is a collection. A stack of newspapers is a collection. But the ways in which these collections are created and operate differ. An insert operation, in the case of the bus queue, must add a person to the end of the line or there may be a riot. Insert in the shopping basket means throw in another grocery item at random: position has no importance (unless the washing powder is on top of the eggs). Insert in the stack of newspapers means put a fresh one on top (the one on the bottom may never be removed from the stack). Traditionally, programmers of C and other languages (including early C++) had to worry about these ordering details. The facilities of the C++ Standard Library relieve them of this burden. They now 'only' have to learn the myriad facilities of a large and complex library.

Although included as part of the ISO C++ Standard, the Library is not part of the language: it is a set of facilities implemented with the language. The Library is large: the standard reference, *The C++ Standard Library* (0-201-37926-0, Josuttis, Addison-Wesley, 1999) – which I highly recommend – exceeds 800 pages in length and appears to me not to be exhaustive. Our *Made Simple* book, therefore, has no chance of making more than a brief introduction to the Library, presenting some of its more common characteristics in the hope that you will be able to extrapolate and move on to more complex facilities, perhaps using Josuttis or other references.

Standard template library

The STL is a subset of the Standard C++ Library. The Standard Library provides the following facilities.

- Data structures and algorithms (containers, iterators, algorithms, function objects, allocators, adapters)
- Stream input and output (the facilities referred to in Chapter 13 as the IOStream Library)
- Strings
- Internationalisation
- Date and time
- Numeric analysis
- Exception hierarchy
- Complex numbers

Of these, the STL comprises containers, iterators, algorithms, adapters, function objects and allocators. In English, containers are like the collections I refer to above: lists, queues, shopping baskets full of groceries, and so on. Iterators are the means of selection from a container: from the front please when the bus stops, grocery items at random by the checkout assistant. Algorithms are ready-made operations that you can carry out on the containers, things like find, search, count and sort. Remember that the great benefit of the whole library arrangement is that you can do these operations without regard for the underlying details of how the stack, queue, or whatever, operates.

Non-STL members of the Standard Library include IOStreams and Strings. And that's the limit of the scope attempted by this book in its consideration of the Library as a whole. Chapter 2 looks briefly at the essential characteristics of the String class. There's a bit more later in this chapter. Chapter 13 sets out the essentials of IOStreams. The next section looks at the STL and its constituent data structures.

Before we do that, you may remember that, to use a given library function (say, from Chapter 3, getline) you're supposed to #include the corresponding header file: iostream in the case of getline. To use all the facilities provided by the Standard Library and the STL, there are many Standard header files that you should be aware of. All told, there are 50: 32 Standard C++ header files, and 18 for the Standard C Library. The C header files have the same names as their C-language predecessors, except that they are prefixed with the letter 'c' and the trailing '.h' is dropped. Thus the old string.h for C-strings becomes cstring.

Here is the full set of C++ header files.

```
<algorithm>   <bitset>       <complex>    <deque>     <exception>
<fstream>     <functional>   <iomanip>    <ios>       <iosfwd>
<iostream>    <istream>      <iterator>   <limits>    <list>
<locale>      <map>          <memory>     <new>       <numeric>
```

<ostream> <queue> <set> <sstream> <stack>
<stdexcept> <streambuf> <string> <typeinfo> <utility>
<valarray> <vector>

followed by the renamed C headers

<cassert> <cctype> <cerrno> <cfloat> <ciso646>
<climits> <clocale> <cmath> <csetjmp> <csignal>
<cstdarg> <cstddef> <cstdio> <cstdlib> <cstring>
<ctime> <cwchar> <cwctype>

I've given the full list here for completeness, but it is beyond the scope of this book – possibly any book – to give examples of all of them in use. The most commonly-used – and the ones that this book concentrates on – are iostream and string, with fstream, iomanip and some of the C headers also appearing occasionally.

For further reading, I recommend *The C++ Standard Library* (Josuttis), (for details of the C headers) *C: A Reference Manual* (0-13-326224, Harbison & Steele, Prentice-Hall 1995) and my own *Newnes C Pocket Book* (0-7506-2538-4).

STL containers

The Standard Template Library provides a range of template based containers. There are two main classifications: sequence containers and associative containers. Sequence containers are ordered collections in which every element has a position. Our bus-queue is a good example. Associative containers are sorted collections in which the position of a given element depends on its value. Think of a telephone directory, where the elements are name-phone-number pairs.

There are three kinds of sequence container: the vector (similar to a C-style array), the *deque* (double-ended queue, like the bus queue) and the *list* (doubly linked, the classical alternative to the array). An array is a contiguous set of memory 'slots', where each 'slot' contains an element. Access is very fast – add one to an index to go to the next 'slot'. However, adding and deleting elements is inefficient: to add or delete in the middle of an array, you have to 'shuffle' other elements right or left. A list, by contrast, is a series of elements linked by pointers. Access may be a bit slower than in the case of arrays because of having to follow chains of pointers, but addition and deletion are done by manipulating pointers and are very fast.

There are two kinds of associative container: the *set* and the *map*. A simple sorted list of names is a set; our telephone directory is a map, where the name is the key and the phone number is the value. Variations on the set and map are the *multiset* and *multimap*; the only difference is that the multiset and multimap allow duplicate entries. The telephone directory is therefore a map but not a multimap: no name-number key-value pair can appear more than once.

The STL also provides three special sequence containers, known formally as *container adapters*. These are for particularly commonly-occurring data structures and are the *stack*, the *queue* and the *priority queue*.

The STL implements all these containers with templates. As far as possible, you, the programmer, are spared having to know how the containers are implemented in detail. You are given a standard set of access mechanisms – *iterators* and *algorithms* – and you can to a great extent treat the different containers in a uniform manner. The obvious benefit is that you don't have to 'reinvent the wheel' by figuring out how to implement a linked list or a binary tree. The details are done for you. The price is that you have to be aware of the facilities of the STL and how to use them.

Vector container

To cover all the features and operations of any of the containers would take several hundred pages. What I do here is simply to introduce use of the vector container, show a number of simple example programs and then present a parallel coverage of the list container, noting differences where they exist. This isn't in any way a comprehensive approach, but it will get you over the 'hump' of basic use of STL facilities, and thereby allow you to experiment further yourself.

Here's the most basic use of vector, shown in the program vectint.cpp:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> vectint;
    for (int i=0; i<10; i++)
        vectint.push_back(i);
    for (unsigned i=0; i<vectint.size(); ++i)
        cout << vectint[i] << endl;
}
```

This uses the member function push_back of the vector template to "push" ten integers into a vector (array) of integers. Then the member function size is used to control a loop that traverses the array one element at a time, displaying the contents. Note that the header file <vector> must be #included. The displayed output is this:

```
0
1
2
3
4
5
6
7
8
9
```

Use of the simple increment mechanism:

```
for (unsigned i=0; i<vectint.size(); ++i)
```

is intuitive but, as we'll see later with the list collection, not portable between collections. A better approach is to use the facility designed for exactly this purpose, the iterator. It's shown in action in the program vectit1.cpp:

```
/*
 *  "vectit1.cpp": vector with simple iterator and reverse iterator
 */
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vectint;
    for (int i=0; i<10; i++)
        vectInt.push_back(i);
    for (unsigned i=0; i<vectInt.size(); ++i)
        cout << vectInt[i] << ' ';
    cout << endl;
    // basic use of iterators over a vector
    vector<int>::iterator beg;
    vector<int>::iterator end;
    vector<int>::iterator pos;
    for (pos=beg=vectInt.begin(), end=vectInt.end(); pos!=end; ++pos)
        cout << *pos << ' ';
    cout << endl;
    // basic use of reverse iterators over a vector
    vector<int>::reverse_iterator rbeg;
    vector<int>::reverse_iterator rend;
    vector<int>::reverse_iterator rpos;
    for (rpos=rbeg=vectInt.rbegin(), rend=vectInt.rend(); rpos!=rend; ++rpos)
        cout << *rpos << ' ';
    cout << endl;
}
```

The essence of this is the definition of three iterator variables, beg, end and pos

```
vector<int>::iterator beg;
vector<int>::iterator end;
vector<int>::iterator pos;
```

These are then used to *iterate over* the vector after being initialised by means of calls to the vector functions begin and end. There is a corresponding *reverse iterator* for going backwards through the array (vector) of integers. The displayed output of the program is thus

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

A range of operations is provided for use on vectors: the program vectop1.cpp illustrates many of them

```
/*
 *  'vectop1.cpp': non-modifying and access operations
 */
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vectInt;
    for (int i=0; i<10; i++)
        vectint.push_back(i);
    for (unsigned i=0; i<vectint.size(); ++i)
        cout << vectint[i] << ' ';
    cout << endl;
    cout << "Vector max size is " << vectint.max_size() << endl;
    cout << "Vector capacity is " << vectint.capacity() << endl;
    vectint.reserve(512);
    cout << "Vector capacity is " << vectint.capacity() << endl;
    cout << "Element at pos 5: " << vectint.at(5) << endl;
    cout << "Element at pos 7: " << vectint[7] << endl;
    cout << "First element: " << vectint.front() << endl;
    cout << "Last element: " << vectint.back() << endl;
    if (vectint.size() == 0 || vectint.empty())
        cout << "Vector is empty" << endl;
}
```

Most of these functions are self-explanatory. The function reserve allocates (in this case) 512 bytes of memory for the vector, thereby changing the capacity of the vector, as found by the function capacity. Here's the output:

```
0 1 2 3 4 5 6 7 8 9
Vector max size is 1073741823
Vector capacity is 256
Vector capacity is 512
Element at pos 5: 5
Element at pos 7: 7
First element: 0
Last element: 9
```

The last of the program examples, vectcst1.cpp, shows several different ways in which vectors can be created and initialised.

```
/*
 *  'vectcst1.cpp': vector constructors & destructor
 */
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1;
    for (int i=0; i<10; i++)
        v1.push_back(i);
    vector<int>::iterator beg=v1.begin();
    vector<int>::iterator end=v1.end();
    vector<int>::const_iterator pos=beg;
```

```
        while(pos!=end)
            cout << *pos++ << ' ';
        cout << endl
        vector<int> v2(v1);
        for (pos=beg=v2.begin(), end=v2.end(); pos!=end; pos++)
            cout << *pos << ' ';
        cout << endl

        beg=v1.begin();
        end=v1.end();
        vector<int> v3(beg,end);
        for (pos=beg=v3.begin(), end=v3.end(), pos!=end; pos++)
            cout << *pos << ' ';
        cout << endl
        vector<int> v4(10,7);
        v4.push_back(11);
        for (pos=beg=v4.begin(), end=v4.end(); pos!=end; pos++)
            cout << *pos << ' ';
        cout << endl;
    }
```

Four vectors are created, using different constructors.

```
vector<int> v1
vector<int> v2(v1);
vector<int> v3(beg,end).
vector<int> v4(10,7).
```

The first creates an instance v1 of a vector of ints. This is uninitialised; the function **push_back** is used in the loop following to put some values into the array elements. The second definition creates the instance v2 initialised to the contents of v1. The instance v3 is initialised with all the values between **beg** and **end** inclusive of both. Finally the vector v4 has all ten elements set to the number 7. This is the displayed output

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
7 7 7 7 7 7 7 7 7 7 11
```

List container

The interface provided by the STL for the programmer to each of the container types includes a range of functions and iterators. These are similar across all the containers but not identical. This is because certain aspects of the layout of the various containers are fundamentally different. For example, with an array (vector), you can go directly to item number 7 using a subscript; it is possible to perform *direct access* on a vector. It is in the nature of linked lists that you can't do this. You must start from the top of the list – or some other place in the list to which a pointer is available – and move along the links from there to the element required. Direct access is not possible with a linked list. In this case, as in others, the interface presented by the STL to the vector container differs from that of the list.

The first place we see the difference is in the simplest of the four programs.

listint.cpp

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
     list<int> listint;
     for (int i=0; i<10; i++)
          listint.push_back(i);
     for (unsigned i=0; i<listint.size(); ++i)
          cout << listint[i] << endl;
}
```

This actually gives a compilation error. You're told that the addition operator is not supported for the list vector. This is where we need the uniform interface provided by the iterator mechanism. The program listit1.cpp is analogous to vectit1.cpp above:

```
/*
 *   "listit1.cpp": list with simple iterator and reverse iterator
 */
#include <iostream>
#include <list>
using namespace std;
int main()
{
     list<int> listint;
     for (int i=0; i<10; i++)
          listint.push_back(i);
//   No random access to list, overloaded + not supported
//   for (unsigned i=0; i<listint.size(); ++i)
//        cout << listint[i] << ' ';
//   cout << endl;
     // basic use of iterators over a list
     list<int>::iterator beg;
     list<int>::iterator end;
     list<int>::iterator pos;
```

```
    for (pos=beg=listint.begin(), end=listint.end(); pos!=end; ++pos)
        cout << *pos << ' ';
    cout << endl;
    // basic use of reverse iterators over a list
    list<int>::reverse_iterator rbeg;
    list<int>::reverse_iterator rend;
    list<int>::reverse_iterator rpos;
    for (rpos=rbeg=listint.rbegin(), rend=listint.rend(); rpos!=rend; ++rpos)
        cout << *rpos << ' ';
    cout << endl;
}
```

The direct-access loop is commented out, while the form of the iterators used in the rest of the program exactly parallels that used for vectors. The displayed output is

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

The range of functions available for lists is smaller than that for vectors, as the equivalent program to vectop1.cpp, listop1.cpp, shows:

```
/*
 *  'listop1.cpp': non-modifying and access operations
 */
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> listint;
    for (int i=0; i<10; i++)
        listint.push_back(i);
//  for (unsigned i=0; i<listint.size(); ++i)
//      cout << listint[i] << ' ';
//  cout << endl;
    cout << "list max size is " << listint.max_size() << endl;
//  cout << "list capacity is " << listint.capacity() << endl;
//  listint.reserve(512);
//  cout << "list capacity is " << listint.capacity() << endl;
//  cout << "Element at pos 5: " << listint.at(5) << endl;
//  cout << "Element at pos 7: " << listint[7] << endl;
    cout << "First element: " << listint.front() << endl;
    cout << "Last element: " << listint.back() << endl;
    if (listint.size() == 0 || listint.empty())
        cout << "list is empty" << endl;
}
```

The functions commented out are not supported by the **list** container. Lastly, we can see from the program listcst1.cpp that the constructors available for creating list instances are similar to their vector counterparts.

```
/*
 *  'listcst1.cpp': list constructors & destructor
 */
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> list1;
    for (int i=0; i<10; i++)
        list1.push_back(i);
    list<int>::iterator beg=list1.begin();
    list<int>::iterator end=list1.end();
    list<int>::const_iterator pos=beg;
    while(pos!=end)
        cout << *pos++ << ' ';
    cout << endl;
    list<int> list2(list1);
    for (pos=beg=list2.begin(), end=list2.end(); pos!=end; pos++)
        cout << *pos << ' ';
    cout << endl;
    beg=list1.begin();
    end=list1.end();
    list<int> list3(beg,end);
    for (pos=beg=list3.begin(), end=list3.end(); pos!=end; pos++)
        cout << *pos << ' ';
    cout << endl;

    list<int> list4(10,7);
    list4.push_back(11);
    for (pos=beg=list4.begin(), end=list4.end(); pos!=end; pos++)
        cout << *pos << ' ';
    cout << endl;
}
```

The program makes the following display:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
7 7 7 7 7 7 7 7 7 7 11
```

The string class

Constructors

Chapter 2 introduces basic use of the C++ Standard Library string class. This section looks at the different string class constructors and ways of creating and initialising string variables. It also gives a summary of available string member functions.

To start, here's a program, constrs.cpp, that shows by example all the string constructors except the ones involving STL iterators.

```
/*
 *   constrs.cpp  exercise all 'string' class constructor
 *   overloadings except the one taking iterator arguments
 */
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s1;                    // default constructor
    s1 = "schadenfreude";
    cout << "String 1: " << s1 << endl;
    string s2(s1);                // copy constructor
    cout << "String 2: " << s2 << endl;
    string s3(s1, 7);             // construct from pos 7
    cout << "String 3: " << s3 << endl;

    // construct from pos 2 through 4
    string s4(s1,2,3);
    cout << "String 4: " << s4 << endl;

    // define C string & character array
    char nulltermstr[30] = "Null-terminated string";
    char chararray[30] = {'c','h','a','r','a','r','r','a','y'};

    // construct from C string
    string s5(nulltermstr);
    cout << "String 5: " << s5 << endl;

    // construct from a number of characters in the array
    string s6(chararray,7);
    cout << "String 6: " << s6 << endl;

    // construct with 10 copies of 'q'
    string s7(10, 'q');
    cout << "String 7: " << s7 << endl;
```

```
        // call destructor for all string instances
        s1.~string();
        s2.~string();
        s3.~string();
        s4.~string();
        s5.~string();
        s6.~string();
        s7.~string();
}
```

The variable s1 is assigned the German word *schadenfreude* – suitable, in the context, I think – using the overloaded assignment operator of **string**. The variable s2 is initialised with the **string** copy constructor; creation of the other five variables causes calls to be made on the remaining **string** constructors. The program is best explained by its output, which shows the results of the creation of **string** instances **s1** to **s7**:

```
String 1: schadenfreude
String 2: schadenfreude
String 3: freude
String 4: had
String 5: Null-terminated string
String 6: chararr
String 7: qqqqqqqqqq
```

There is one more way of creating an instance of **string**: with STL iterators. This is inadequately explained in other references. I hope the two programs that follow make the mechanism clear. First, **iterator.cpp**:

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
        vector<char> str1;
        str1.push_back('n');
        str1.push_back('o');
        str1.push_back('p');
        str1.push_back('q');
        str1.push_back('s');
        str1.push_back('t');
        str1.push_back('u');
        str1.push_back('v');
        str1.push_back('w');
        // set vector position half-way, insert before 's'
        for(tmp=beg; tmp!=end && *tmp!='s'; tmp++)
            ;

        if(tmp!=end)
            str1.insert(tmp,3,'y');

    // 'v', 'w', 'x' fall off the right end
        string s2(beg,end);
        cout << "string 2: " << s2 << endl;

        // now re-take begin and end, 'v', 'w' and 'x' reappear
        beg = str1.begin();
        end = str1.end();
        string s3(beg,end);
        cout << "string 3: " << s3 << endl;
    }
```

This builds on iterator.cpp. Three iterators, beg, tmp and end, are used to insert the character 'y' three times halfway along the char vector str1. This is then used to create and initialise the string instance s2. Three characters 'fall off the end'; enough space for all the characters of the expanded string is allocated by creation of the string instance s3. Here's the output:

```
string 1: nopqstuvwx
string 2: nopqyyystu
string 3: nopqyyystuvwx
```

Member functions

Following is a table listing the member functions of the Standard C++ string class, along with a short description of how to use each.

Function	Prototype	Usage

length, size	size_type length() const
	(size_type is an unsigned integer)
		string str = "abcdef"
		str.length() // == 6
		str.size() // == 6

insert	string& insert
	(size_type pos, const string& str);
		string str1 = "abcdef";
		string str2 = "xxx";
		str1.insert(4, str2);
		// str1 now "abcdxxxef"

erase	string& erase
	(size_type pos=0, size_type len),
		// Delete a substring.
		// starting after position
		// pos for the length len
		str1.erase(4 3),
		// str1 now "abcdef"

find, rfind	size_type find
	(const string& str, size_type pos=0) const.
	size_type find
	(char ch, size_type pos=0) const;
	size_type rfind
	(const string& str, size_type pos) const;
	size_type rfind
	(char ch, size_type pos) const;
		// Search for the first
		// occurrence of the
		// substring str (or
		// character ch) in the
		// current string, starting at
		// pos rfind returns last
		// occurrence
		str1 find("cde", 0);
		// returns 2

replace	string& replace
	(size_type pos, size_type n, const string &s);
		/ Delete a substring from
		the current string and
		// replace with another
		string str1 = "abcdef",
		string str2 = "xxx";
		str1 replace(3,2, str2);
		// str1 now "abcxxxf"

substr	string substr
	(size_type pos, size_type n) const,
		// Return a substring of the
		// current string, starting at
		// pos for length n
		str2 = str1 substr(3 3);
		// str2 now "xxx"

find_first_of
find_last_of
find_first_not_of
find_last_not_of
	size_type find_first_of
	(const string& str, size_type pos=0) const:
	size_type find_last_of
	(const string& str, size_type pos) const;
	size_type find_first_not_of
	(const string& str, size_type pos=0) const:
	size_type find_last_not_of
	(const string& str, size_type pos) const;
		Search for the first/last
		// occurrence of a
		// character that does/does
		// not appear in str
		string str1 = "abcdef",
		string str2 = "xxbbzz"
		str1.find_first_of(str2,0),
		/ returns 2

c_str	const char* c_str() const:	// Convert an instance of
		// string to a C-string
		string str1 = "abcdef"
		char charray[20];
		const char* cstr = charray
		cstr = str1.c_str(),
		// cstr now "abcdef"

characters (of two or more bytes) as well as the 'normal' one-byte characters used in the English speaking world. The ISO standards committee had two options: build a library for 'normal' characters and then duplicate that library for wide characters, or use C++ templates in the manner intended, parameterising the particular type of character being handled and instantiating classes accordingly. The committee chose the template approach.

The base class ios_base is just a class, not a template. It defines properties such as format and state flags for all I/O stream classes — standard input, standard output, files and others. The ios and wios classes instantiated from the derived template basic_ios<> define properties of all stream classes that depend on the distinction between 'normal' and wide characters. Actual input to and output from any stream is done by low-level operations defined by the template basic_streambuf<> and instantiated by either of the classes streambuf or wstreambuf. The templates basic_istream<> and basic_ostream<> and their instantiations istream, ostream, wistream and wostream are those most commonly used by C++ programmers. They are typically used to direct text output to the standard output device or to receive text input from standard input. They rely on all the low-level services provided by the classes and templates from which they are derived. The classes ifstream, ofstream and fstream are respectively for input, output and input-output operations on disk files.

Throughout this book you've seen instances of the istream and ostream classes. Along with their wide-character counterparts, these *global stream objects* are summarised in this table:

Type	Name	Purpose
istream	cin	Standard input
ostream	cout	Standard output
ostream	cerr	Standard error
ostream	clog	Buffered cerr for log output
wistream	wcin	Standard input for wide characters
wostream	wcout	Standard output for wide characters
wostream	wcerr	Standard error for wide characters
wostream	wclog	Buffered cerr for wide-character logs

The IOStream library class hierarchy is declared in header files including istream, ostream, iostream, streambuf, fstream and iomanip. Inside these files, exactly where and how things are declared is implementation-dependent. The internals of the header files will differ between, say, Microsoft C++ and Borland C++, but the available facilities should be the same from the programmer's standpoint.

The operators << for output and >> for input are overloaded. The << operator overloaded for stream output is called the insertion operator or inserter. When used, it is said to insert bytes into the output stream. The >> operator overloaded for stream input is called the extraction operator or extractor. When used, it is said to extract bytes from the input stream.

The extractor and inserter operators are basic C++ bit shift operators overloaded to have multiple definitions as operator functions. The multiple overloadings allow you to use the operators for input and output of objects of many different data types: they, not you, take care of type-safety.

Type safety is one of the benefits of the IOStream library and of C++ in general: the interface presented to the programmer does not change with data type. Suppose we have a data object X of one of the types char, int, float and double, but we don't know which it is, then the statement:

```
cout << X;
```

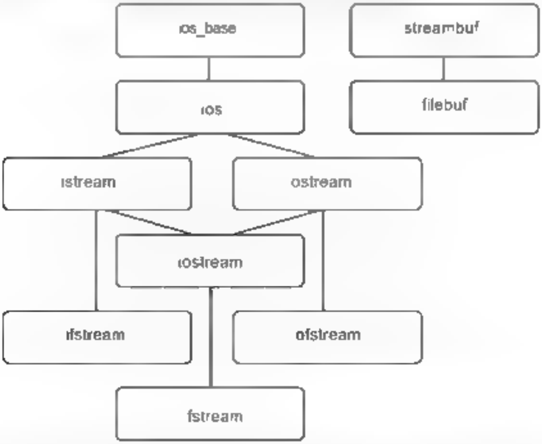
using the IOStream operator << correctly sends to the output stream the value of X. On the other hand, the C statement:

```
printf("%d", X);
```

fails if X is, for instance, of type double. There's no easy way in C of implementing functions that operate correctly on arbitrary types. IOStream library functions and operators do this in a type-safe manner by taking advantage of the C++ function- and operator-overloading capabilities.

The IOStream library classes

In keeping with the *Made Simple* nature of this book, the wide character variant of the IOStream library is not dealt with further here. The remainder of this chapter is confined to consideration of C++ Stream I/O using only the one-byte characters "normal" in the English-speaking world. This allows me to present this simplification of the IOStream library hierarchy diagram as the basis of what follows.



Everything is this diagram is a class. The **streambuf** class is not part of the main hierarchy: it is referenced by a pointer defined in the class ios whenever its low-level input/output capabilities are required. **streambuf** allocates memory for and maintains the stream buffer objects. Usually, you don't have to be concerned with the definitions contained in the **streambuf** class or the details of the low-level I/O performed by it.

The **ios_base** contains more information about the state of the stream. This includes the stream open mode, seek direction and format flags. Along with format flags, the following functions are also members of the **ios_base** class:

flags **setf** **unsetf** **width** **fill** **precision**
tie **rdstate** **eof** **fail** **bad** **good** **clear**

The functions in the first row are for data formatting and are explained in the next section.

The ios class, derived from ios_base, defines a pointer to an instance of streambuf. From ios are derived the input stream istream and output stream ostream classes. These declare I/O functions and operators that are used by the C++ programmer. Both istream and ostream are #included in programs with the header file iostream. This header file in turn #includes the header files istream and ostream, where the classes of the same name are declared. istream contains function declarations including the following:

get
read
getline
gcount
peek
putback
seekg
tellg

The class ostream includes these function declarations:

put
write
seekp
tellp

istream also contains the definitions of the overloaded extractor >>, while ostream contains the definitions for the inserter <<.

The class iostream inherits both istream and ostream. It is declared in the iostream header file. The file I/O classes ifstream and ofstream are declared in the header file fstream. ifstream inherits all the standard input stream operations defined by istream and adds a few more, such as constructors and functions for opening files. ofstream similarly augments the inherited definitions of ostream.

Finally, fstream, declared in the header file fstream, inherits iostream and contains functions and constructors that allow files to be opened in input/output mode.

Formatted I/O

All Stream I/O done in earlier chapters is unformatted: the formats of output and input data are default settings used by the insertion and extraction operators. You can, in three ways, specify these formats explicitly:

- The setf, unsetf and flags functions use format flags to alter input and output data. The ios_base class enumerates the flag values and also declares the functions.
- The ios_base class member functions width, precision and fill are used to set the format of input and output data.
- Using manipulators, special functions that combine the above two techniques and add some more.

Format flags

Every C++ input and output stream has defined for it (in the ios_base class) a number of format flags that determine the appearance of input and output data. These flags represent different patterns of bits stored within a long integer like this:

```
// skip white space on input
skipws   = 0x0001,
// left-adjust output
left     = 0x0002,
// right-adjust output
right    = 0x0004,
```

This is just a likely sample but should not be taken as literally true for every C++ environment. The exact setting of each of the format flags is implementation-dependent. This is the full set of format flags:

```
dec        // decimal conversion, mask ios_base::basefield
oct        // octal conversion, mask ios_base::basefield
hex        // hexadecimal conversion, mask ios_base::basefield
left       // left-adjust output, mask ios_base::adjustfield
right      // right-adjust output, mask ios_base::adjustfield
internal   // left-adjust sign, right-adjust value
           // mask ios_base::adjustfield
scientific // scientific notation, mask ios_base::floatfield
fixed      // decimal notation, mask ios_base::floatfield
skipws     // skip white space on input
showbase   // show integer base on output
showpoint  // show decimal point
uppercase  // uppercase hex output
showpos    // explicit + with positive integers
unitbuf    // flush output after each output operation
```

You can see that some of the flags are associated with masks. These are for use with the second overloaded version of the setf function, one of several ios_base member functions used to manipulate output and input formats. In general, the effect of the masks is to clear all current format settings for the mask's group before applying new settings. The mask ios_base::basefield is for the flag group that contains the flags dec, oct and hex. You can see the other two masks and their flag groups in the list above.

Here's a simple example program, format02.cpp, of how to use format flags. We define an integer and initialise it to a decimal value. We then write it to the standard output in its hexadecimal form, showing the base (0X) in uppercase.

```
#include <iostream>
using namespace std;

int main()
{
    int number = 45;

    // set hexadecimal and show the base
    cout.setf(ios_base::hex, ios_base::basefield);

    cout.setf(ios_base::showbase | ios_base::uppercase);
    cout << number << endl;
}
```

The output of the code is this:

```
0X2D
```

Both overloaded versions of the setf function are used here. The first use of setf is the version taking two parameters. The mask ios_base::basefield clears any decimal, octal or hexadecimal flags that may already be set and then sets the hexadecimal flag. The second setf takes a single parameter, a *bitwise OR* combination of the showbase and uppercase flags. The combined effect of the two setfs is to format the decimal number 45 to uppercase hexadecimal with the base ('X') explicitly shown.

The format flags set in this way cause all subsequent integers written to the standard output to be displayed in hexadecimal until the flags are changed or unset. You switch the flags off using the unsetf function:

```
cout.unsetf(ios_base::hex | ios_base::showbase | ios_base::uppercase);
```

After this operation, the output format reverts to what it was before the call to setf: the decimal number 45. Here are the prototypes of the setf, unsetf and flags functions:

```
ios_base::fmtflags setf(ios_base::fmtflags);
ios_base::fmtflags setf(ios_base::fmtflags, long);
ios_base::fmtflags unsetf(ios_base::fmtflags);
```

```
ios_base::fmtflags flags();
ios_base::fmtflags flags(ios_base::fmtflags);
```

The function setf called with a single argument of type ios_base::fmtflags (similar to a long integer) turns on the format flags specified in that argument. setf returns the format flag values as they were before the call to setf. An overloaded definition of setf takes two arguments. A call to this function turns off the flags specified by the second argument (the mask) and then turns on the flags specified by the first. The function returns the format flag values as they were before it was called.

The function unsetf turns off the flags specified by its argument and returns the format flag values as they were before unsetf was called.

A call to the flags function without arguments returns the current state of the format flags. The flags function called with one long argument sets the format flags to the values specified by that argument and returns the values of the flags as they were before the call to flags. This function is important in being the only one of the five shown that actually clears all previous format flag settings.

Manipulating format flags

Here's an example program, format1.cpp, that exercises all five flag-setting functions as well as a number of the format flags.

```
#include <iostream>
using namespace std;

int main()
{
    ios_base::fmtflags old_flags;    // old flag values
    ios_base::fmtflags tmp_flags;    // temporary flag values
    ios_base::fmtflags new_flags;    // new flag values
    int number = 45;

    //  store original format flag values
    old_flags = cout.flags();

    //  show + sign if positive
    cout.setf(ios_base::showpos);
    cout << number << endl;

    //  set uppercase hexadecimal and show the base
    tmp_flags = cout.setf(ios_base::hex, ios_base::basefield);

    //  following 3 statements have the same aggregate effect as previous setf
    //tmp_flags = cout.flags();
    //cout.unsetf(ios_base::dec);
    //cout.setf(ios_base::hex);
    cout.setf(ios_base::showbase | ios_base::uppercase);
    cout << number << endl;

    //  display twice to show that setf is persistent
    cout << number << endl
    //  unset the uppercase flag
    cout.unsetf(ios::uppercase);
    cout << number << endl;

    //  revert to showpos only
    cout.setf(tmp_flags);
    cout << number << endl;

    //  return to original format flag values
    new_flags = cout.flags(old_flags);
    cout << number << endl;
}
```

This is the output when the program is run.

```
+45
0X2D
0X2D
0x2d
+45
45
```

This program is worth careful reading, testing and experimentation, as it weeds out and clarifies many of the subtleties of the formatting flags and functions. In particular, examine these lines:

```
tmp_flags = cout.setf(ios_base::hex, ios_base::basefield)

//  following 3 statements have the same aggregate effect as previous setf
//tmp_flags = cout.flags();
//cout.unsetf(ios_base::dec);
//cout.setf(ios_base::hex);
```

Because the individual flag values are members of the ios_base class, they must be scope resolved when they are used: ios_base::hex is correct, while hex alone is not. In older, pre-Standard C++ versions, the flags used be members of the class ios, not ios_base. Most Standard-conforming C++ environments still allow you to use, for example, ios::hex.

These are the characteristics of the other format flags.

- dec is used to control the number base, converting output integers to decimal and causing input integers to be treated as decimal; it is the default base value.
- skipws, if set (which is the default), causes white space to be skipped on input using the extraction operator.

The class **ostream** contains a member function **flush** with this prototype

```
ostream& flush();
```

Using flush has the effect of flushing and writing the contents of the buffered stream for which it is called.

```
cout.flush(); // function call
```

Stream input

The facilities provided by Stream I/O for input are symmetrical to those for output. The overloaded extraction operator >> is used for stream input. Extractors share many characteristics with inserters:

- Extractors, like inserters, are overloaded operator functions.
- You can chain extraction operations in the same way as insertion operations.
- You can use an extractor on any input stream; there is no concept of separate operations for different streams along the lines of scanf and fscanf in the C Standard Library.
- As with inserters, you can customise your own extractors. And again, see the *C++ User's Handbook* or the *C++ Programming Language* (3rd edn).

The following are the built-in Stream I/O inserter types

```
char (signed and unsigned)
short (signed and unsigned)
int  (signed and unsigned)
long (signed and unsigned)
char * (string)
float
double
long double
```

Extraction operations that accept input from the standard I/O stream cin by default skip leading white spaces and white space-separated input. This can be changed with the skipws format flag or the ws manipulator (see the example manip7.cpp in the section on *Input manipulators*, page 314). Extraction fails if data of a type not matching the receiving variables is received.

Functions

Stream I/O provides a number of functions, in addition to the inserters already described, for simple input from a stream. The get function has a number of overloadings allowing different definitions of the function to perform different tasks on an input stream. These are the get prototypes:

```
int      get();
istream& get(char&);
```

operator >> is OK where text input is continuous, but input stops as soon as the first white space character is seen. If you want multi-word text, you've got to use either get or getline. Depending on the type of data you are using, one or the other may be suitable. Here's the first illustrative program, get.cpp

```
#include <iostream>
using namespace std;
const int MAX = 50;

int main()
{
    char    str1[MAX];
    char    *s1 = str1;
    cout << "Enter string: ";
    while (cin.get(s1, MAX))
    {
        cin.get();
        cout << s1 << endl;
        cout << "Enter string: ";
    }
}
```

The while loop terminates when the call to cin.get returns end-of-file (EOF). This is generated from the keyboard by pressing *Ctrl Z*. This code, when run, accepts lines of non continuous text, each up to 50 characters long, until you press EOF. If, however, you use getline in place of get, you may find that the RETURN key must be pressed multiple times after each entered line to get a new prompt. Text input to an ordinary C-string seems to be best done with cin.get

Where the ISO C++ string class is used, getline works well, as shown by the program getline.cpp.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;
    cout << "Enter String: ";
    while (getline(cin, line))
    {
        cout << line << endl;
        cout << "Enter String: ";
    }
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char string[20] = {'a','b','c','d','e','f'};
    double pi = 3.141592654;
    int n_dec = 35;
    int n_oct = 035;
    cout.setf(ios_base::fixed, ios_base::floatfield);

    //  Demonstrate simple output manipulators
    cout << "Octal: " << oct << n_dec << endl;
    cout << "Decimal: " << dec << n_oct << endl;
    cout << "Hex: " << hex << n_oct << endl;

    //  Rightmost manipulator overrides others
    cout << "Hex: " << hex << dec << n_oct << endl;

    //  Convert octal number to decimal, pad output field of width 6 with blanks
    cout << "Padded: " << setw(6) << dec << n_oct << endl;

    //  Equivalent operation: convert using setbase and pad field with zeros
    cout << "Padded: " << setw(6)
            << setfill('0')
            << setbase(10)
            << n_oct
            << endl;

    //  Pi output in field-width 8, precision 4
    cout << "Rounded Pi: " << setw(8)
            << setprecision(4)
            << pi
            << endl;

    //  Precision 8, field width 4: output is expanded
    cout << "Rounded Pi: " << setw(4)
            << setprecision(8)
            << pi
            << endl;

    //  Output null-terminated character array
    cout << "String: " << string << ends << endl;

    //  Display Pi in scientific notation
    //  Explicitly unset fixed first
    cout.unsetf(ios_base::fixed);
```

```
        cout << "Exponent PI: "
                << setiosflags(ios_base::scientific)
                << pi << endl;

        //  Display an integer left-justified in hex
        //  Unset decimal first
        cout.unsetf(ios_base::dec);
        cout << "Hex: " << setw(10)
                << setiosflags(ios_base::left | ios_base::hex)
                << n_dec << endl;

        //  Display an integer right-justified in hex
        cout << "Hex: " << setw(10)
                << resetiosflags(ios_base::left)
                << n_dec << endl;

        //  Flush output and stop
        cout << "Finished..." << flush << endl;
}
```

The output displayed by the program is this:

```
Octal: 43
Decimal: 29
Hex: 1d
Hex: 29
Padded:     29
Padded: 000029
Rounded PI: 003.1416
Rounded PI: 3.14159265
String: abcdef
Exponent PI: 3.14159265e+00
Hex: 2300000000
Hex: 0000000023
Finished...
```

The manipulators setiosflags and resetiosflags, combined in use with the format flags defined in the class ios_base, are equivalent to setf and unsetf, while promoting shorter and more concise coding. The include file iomanip must be included if manipulators taking arguments are used.

Surprises in programs such as manip1.cpp are caused mainly by failure to turn off flags using masks. To do fixed point formatted output, explicitly set fixed, turning off all floating point flag bits:

```
cout.setf(ios_base::fixed, ios_base::floatfield);
```

Similarly, before setting scientific, fixed is explicitly unset, while before setting hex, dec is unset.

Input manipulators

The IOStreams Library provides a set of built-in manipulators for input. Input manipulators are defined and used in a way that is essentially the reverse of output manipulators. All the rules surrounding use of input manipulators are the same:

- Input manipulators are embedded between extractor operators
- The setiosflags and resetiosflags manipulators combine format flags and manipulators
- setiosflags is equivalent in effect to setf.
- resetiosflags is equivalent in effect to unsetf
- iomanip must be #included if input manipulators are used which take arguments

Here is a list of built-in manipulators for input from any stream

Manipulator	Purpose
dec	decimal conversion (default)
hex	hexadecimal conversion
oct	octal conversion
ws	skip white space characters
resetiosflags(f)	reset format bits specified by f
setfill(c)	set fill character to c
setiosflags(f)	set format bits specified by f
setw(w)	set field width to w

The following example program, manip2.cpp, shows a number of these manipulators in use

```
#include <iostream>
using namespace std;

#include <iomanip>

int main()
{
    //  do a numeric conversion
    int n_dec;

    cout << "Enter a hexadecimal number: ";

    //  dont skip leading white spaces!!
    cin >> resetiosflags(ios_base::skipws) >> hex
            >> n_dec;
```

```
      cin.get();
      cout << "Decimal conversion of hex input: "
              << n_dec << endl;

      //   break an input string
      char buf1[20];
      char buf2[20];

      cout << "Enter a string" << endl;
      cin >> setw(10) >> buf1;
      cin >> buf2;
      cout << "String 1 " << buf1 << endl;
      cout << "String 2 " << buf2 << endl;
}
```

The first interesting aspect of this program is the unsetting of the default flag ios_base::skipws. Ordinarily, white space before input of the hexadecimal number is ignored. The new flag setting causes leading white spaces to be treated as part of the number, with predictably unpleasant results.

The second part of the program accepts an array of characters of arbitrary length from the input stream. If the input contains more than 10 characters, it is broken into two parts. If the input is "abcdefghijklmnopq" then the contents of buf1 are displayed at the end of the program as a null-terminated string of nine characters:

```
abcdefghi
```

The remainder of the characters are stored, null-terminated, in **buf2**.

The **setw** manipulator is useful for ensuring that the length of data input to an array with an extractor does not exceed the array bounds; the data that cannot be accommodated in the array is discarded or used by the next input operation.

Here's the displayed output of **manip2.cpp**, with user input in boldface:

```
Enter a hexadecimal number: 45
Decimal conversion of hex input: 69
Enter a string
abcdefghijklmnopq
String 1 abcdefghi
String 2 jklmnopq
```

File I/O

Using the IOStream library, you open a file by linking it with an input, output or input-output stream. You do this either by explicitly calling the stream member function open or allowing the stream constructor to open the file implicitly. A file is closed by disassociating it from its stream. This is done either explicitly by the stream member function close or implicitly by the stream destructor.

To use files under Stream I/O, you must include the fstream header file. fstream includes the three classes ifstream, ofstream and fstream, for input, output and input-output files respectively. These classes declare all the functions needed to access files in input, output and input/output modes.

Before a file can be opened, you must define an object of the required stream type:

```
ifstream ins;
```

Then you can open the file:

```
ins.open("infile");
```

You can alternatively open the file automatically using the ifstream constructor:

```
ifstream ins("infile");
```

If the file infile does not exist, it is created. When the file has been opened, the stream object ins keeps track of the current state of the file: its size, open mode, access characteristics, current position of the read pointer, and error conditions, if any. You can close the file explicitly using the stream member function close:

```
ins.close();
```

or else rely on the ifstream destructor to close the file when the stream object ins goes out of scope.

It's OK to open a file using its name only, but there are many other options. The full prototype of the input stream open function, declared in the class ifstream, is this:

```
void open(char *n, int m = ios_base::in, int p = filebuf::openprot);
```

The output stream open function, declared in ofstream, has this prototype:

```
void open(char *n, int m = ios_base::out, int p = filebuf::openprot);
```

The input-output open function is declared in fstream as follows:

```
void open(char *n, int m, int p = filebuf::openprot);
```

The first argument in all cases is a string representing the file name, the second is the open mode, and the third, the file access permissions. The open mode for input files is by default ios_base::open. For output files, it is by default ios_base::out.

Let's look at a number of examples of simple file I/O. All the examples are based around the same program, which simply copies one text file to another. To do so, we call a filecopy function.

Basic file copy

Here's the basic program, filecopy.cpp:

```
#include <iostream>

using namespace std;

#include <fstream>
void filecopy(ifstream &, ofstream &);

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        cout << "Invalid arguments specified" << endl;
        return(0);
    }
    ifstream fin(argv[1]);

    if (!fin)
    {
        cout << "Cant open input file" << endl;
        return(0);
    }
    ofstream fout(argv[2]);

    if (!fout)
    {
        cout << "Cant open output file" << endl;
        return(0);
    }

    filecopy(fin, fout);
    fin.close();
    fout.close();
}

//  Function filecopy copies character-by-character from the input to
//  the output stream
void filecopy(ifstream &in, ofstream &out)
{
    char c;
    while (in.get(c), !in.eof())
        out.put(c);
}
```

You can execute the program by entering at the command line

```
filecopy infile outfile
```

The file infile is linked to the input stream ifstream and opened. If for some reason it can't be opened, the stream object fin is set to null, an error is reported and the program stops. If the file outfile can't be opened, the program similarly stops. If both files are successfully opened, their associated stream objects fin and fout are supplied as reference arguments to the function filecopy. This function then reads characters from the input file and writes them to the output file, stopping when end-of-file is encountered on the input file. The error-state function eof, declared in the class ios_base, returns TRUE on end-of-file.

In the next example, we see the files being opened using explicit open function calls. The output file is opened in input output mode and, after the copy, is opened in input mode and displayed. Only the main function is shown.

```
#include <iostream>

using namespace std;

#include <fstream>
void filecopy(ifstream &, fstream &);
int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        cout << "Invalid arguments specified" << endl;
        return(0);
    }
    ifstream fin;
    fin.open(argv[1], ios::in);

    if (!fin)
    {
        cout << "Cant open input file" << endl;
        return(0);
    }
    fstream fout;
    fout.open(argv[2], ios::out);

    if (!fout)
    {
        cout << "Cant open output file" << endl;
        return(0);
    }

    filecopy(fin, fout);
//  now close, open and read the output file
    char c;
    fout.close();
```

```
        fout.open(argv[2], ios::in);
        while (fout.get(c), !fout.eof())
            cout << c;
        fin.close();
        fout.close();
}
```

If we had opened the output file in append mode with

```
fin.open(argv[2], ios::app);
```

or

```
fstream fin(argv[2], ios::app);
```

the contents of the input file would be added to the end of any existing output file instead of overwriting it. In the case of append mode, if the file does not already exist, it is created.

We can implement the filecopy function using the multi-character overloading of the get function. Here, I haven't changed the main function from the original filecopy.cpp. Only the filecopy function is shown.

```
const int MAX = 100;

void filecopy(ifstream &in, ofstream &out)
{
    char instring[MAX];

    while (in.get(instring, MAX, '\n'), !in.eof())
    {
        out << instring;

        // get and copy the newline

        char c;
        c = in.peek();
        if (c == '\n')
        {
            in.get(c);
            out.put(c);
        }
    }
}
```

We define a local character array, instring, to act as an input buffer. Then we read a line from the input file up to, but not including, the trailing newline. If we don't find a newline character, we read a maximum of 100 characters. Either way, the characters are stored in instring, the contents of which are then written to the output file using the built-in inserter.

At the end of each line, the newline must be processed. We do this here a little over-elaborately, the peek function being used to check that the next character is indeed a newline before the copy. We could do the same job in a more crude (and error-prone) way:

```
while (in.get(instring, MAX), !in.eof())
{
    out << instring;

    // get and copy the newline

    in.get();
    out.put('\n');
}
```

In this case, the get call uses the fact that its third argument, the delimiter, has the default value '\n'. We then use the version of the get function that takes no arguments to discard the next character after the line is read. The put function then writes a hard-coded newline character to the output file.

In the final variant of the filecopy function, we use getline to read the input file and gcount to count the characters actually read:

```
const int MAX = 100;

void filecopy(ifstream &in, ofstream &out)
{
    long total_chars = 0;
    char instring[MAX];

    while (in.getline(instring, MAX, '\n'), !in.eof())
    {
        total_chars += in.gcount();
        out << instring;
    }
    cout << "File copied: " << total_chars
            << " bytes" << endl;
}
```

getline reads from the input file a newline-terminated line, including the newline character. The built-in inserter then writes the line to the output stream. On each iteration, we increment the total number of characters actually read. At the end of the function, we report the number of characters copied.

Random file access

Some basic facilities are provided by the IOStream library for random access, that is, starting file access at any point in the file. For portability, you should perform random access operations only on files opened in binary mode. The six functions you're given in Stream I/O to do random access on binary files are these:

read Read a string of characters from an input stream

write Write a specified number of characters to an output stream

seekg Move the position of the file read pointer to a specified offset

tellg Return the current position of the file read pointer

seekp Move the position of the file write pointer to a specified offset

tellp Return the current position of the file write pointer

Once again noting that this is a *Made Simple* book, and having covered *Made Simple* file access, I must here again refer you to the *C++ Users Handbook* for explanation and examples of use of these functions.

#include <cstdio>
void clearerr(FILE *fp);
clearerr clears end-of-file and error status indicators for the file pointed to by fp.

#include <cmath>
double cos(double x);
cos returns the cosine of x in radians.

#include <cmath>
double cosh(double x);
cosh returns the hyperbolic cosine of x.

#include <cstdlib>
div_t div(int n, int d);
div calculates the quotient and remainder of n/d. The results are stored in the int members quot and rem of a structure of type div_t. The type div_t is defined in cstdlib.

#include <cstdlib>
void exit(int status);
exit causes immediate normal program termination. The value of status is returned to the operating system environment. Zero status is treated as indicating normal termination.

#include <cmath>
double exp(double x);
exp returns the value of e raised to the power of x.

#include <cmath>
double fabs(double x);
fabs returns the absolute value of x.

#include <cstdio>
int fclose(FILE *fp);
fclose discards any buffered input or output for the file pointed to by fp and then closes the file. The function returns zero for successful file closure or EOF on error.

#include <cstdio>
int feof(FILE *fp);
feof returns non-zero if the end of the file pointed to by fp has been reached; otherwise zero is returned.

#include <cstdio>
int ferror(FILE *fp);
ferror checks if a file operation has produced an error. It returns non-zero if an error occurred during the last operation on the file pointed to by fp, zero otherwise.

#include <cstdio>
int fflush(FILE *fp);
fflush causes the contents of any buffered but unwritten data to be written to the file pointed to by fp. The function returns zero if successful, EOF on failure.

#include <cstdio>
int fgetc(FILE *fp);
fgetc returns the next character from the file pointed to by fp. It returns EOF on error or end-of-file.

#include <cstdio>
int fgetpos(FILE *fp, fpos_t *ptr);
fgetpos stores in the pointer ptr the current position in the file pointed to by fp. The type fpos_t is defined in cstdio. The function returns non-zero on error.

#include <cstdio>
char *fgets(char *s, int n, FILE *fp);
fgets reads a string from the file pointed to by fp until a newline character is encountered or n - 1 characters have been read. If a newline is encountered it is included in the string s which is null-terminated in any event. The function returns s, or NULL on end-of-file or error.

#include <cmath>
double floor(double x);
floor returns the largest integer, represented as a double, which is not greater than x.

#include <cmath>
double fmod(double x, double y);
fmod returns the remainder of the division of x by y. If y is zero, the result is undefined.

#include <cstdio>

FILE *fopen(const char *s, const char *mode),

fopen opens the file named in the string s in accordance with the open mode specified in the string mode Legal modes are "r", "w" and "a" for reading, writing and appending; any of these suffixed with a + additionally opens the file for reading and writing If a b is suffixed to the mode string a binary file is indicated fopen returns a pointer to the file opened or NULL on error

#include <cstdio>

int fprintf(FILE *fp, const char *<format>, ...),

fprintf is the same as printf, given below, except that its output is written to the file pointed to by fp

#include <cstdio>

int fputc(int c, FILE *fp);

fputc writes the character c to the file pointed to by fp. It returns c, or EOF on error

Although c is defined as an integer it is treated as an unsigned char in that only the low-order byte is used

#include <cstdio>

int fputs(const char *s, FILE *fp),

fputs writes the string s to the file pointed to by fp The function returns a non-negative number or EOF on error

#include <cstdio>

size_t fread(void *buf, size_t n, size_t count, FILE *fp),

fread reads, from the file pointed to by fp into the array at buf, up to count objects of size n. The function returns the number of objects read.

#include <cstdlib>

void free(void *p);

free deallocates the memory pointed to by p and makes it available for other use Before free is called, memory must have been allocated and p initialised by one of the library functions malloc calloc or realloc Equivalent to free, and sometimes more robust, is realloc(p, 0); According to the ISO specifications, free should work but do nothing when p is NULL

#include <cstdio>

FILE *freopen(const char *s, const char *mode, FILE *fp),

freopen opens the file named in the string s and associates with it the file pointer fp The function returns that file pointer or NULL on error

#include <cmath>

double frexp(double x, int *exp),

frexp splits a floating-point number x in two parts a fraction f and an exponent n such that f is either zero or in the range 0.5 and 1.0 and x equals f*(2**n). The fraction is returned and the exponent n stored at exp. If x is initially zero the returned parameters are also both zero

#include <cstdio>

int fscanf(FILE *fp, const char *<format>, ...),

fscanf is the same as scanf, given below, except that the input is read from the file pointed to by fp

#include <cstdio>

int fseek(FILE *fp, long n, int origin);

fseek is usually used with binary streams. When so used, it causes the file position for the file pointed to by fp to be set to a displacement of n characters from origin origin may be any of three macro values defined in cstdio SEEK_SET(start of file), SEEK_CUR(current position in file) or SEEK_END(end-of-file). Used with text streams, n must be zero, or a return value from ftell with origin set to SEEK_SET The function returns non-zero on error

#include <cstdio>

int fsetpos(FILE *fp, const fpos_t *ptr);

fsetpos returns the position of fp to the position stored by fgetpos in ptr The function returns non-zero on error

#include <cstdio>
long ftell(FILE *fp);
ftell returns the current file position for the file pointed to by fp, or returns -1L on error.

#include <cstdio>
size_t fwrite(void *buf, size_t n, size_t count, FILE *fp);
fwrite causes count objects of size n bytes to be written from buf to the file pointed to by fp and returns the number of such objects written. A number less than count is returned on error.

#include <cstdio>
int getc(FILE *fp);
getc reads the next character from the file pointed to by fp and returns it, or EOF on end-of-file or error. getc is a macro and is equivalent to fgetc.

#include <cstdio>
int getchar();
getchar reads the next character from standard input and returns that character, or EOF on end-of-file or error. getchar() is functionally equivalent to getc(stdin).

#include <cstdlib>
char *getenv(const char *s);
getenv returns the operating system environment string associated with the identifier named in the string at s. If no value is associated with the name in s, getenv returns a null pointer. Further details are system-dependent.

#include <cstdio>
char *gets(char *s);
gets reads from standard input an input line into the array at s, replacing the terminating newline with a null terminator. The string s is also returned by gets, or a null pointer on end-of-file or error.

#include <cctype>
int isalnum(int c);
isalnum returns non-zero if c is alphanumeric, zero otherwise.

#include <cctype>
int isalpha(int c);
isalpha returns non-zero if c is alphabetic, zero otherwise.

#include <cctype>
int iscntrl(int c);
iscntrl returns non-zero if c is a control character (0 to 037, or DEL (0177), in the ASCII set), zero otherwise.

#include <cctype>
int isdigit(int c);
isdigit returns non-zero if c is a digit, zero otherwise.

#include <cctype>
int isgraph(int c);
isgraph returns non-zero if c is a printable character other than a space, zero otherwise.

#include <cctype>
int islower(int c);
islower returns non-zero if c is a lowercase letter in the range a to z, zero otherwise.

#include <cctype>
int isprint(int c);
isprint returns non-zero if c is a printable character including space, zero otherwise.

#include <cctype>
int ispunct(int c);
ispunct returns non-zero if c is a printable character other than space, letter and digit, zero otherwise.

#include <cctype>
int isspace(int c);
isspace returns non-zero if c is any of space, tab, vertical tab, carriage return, newline or formfeed, zero otherwise.

#include <cctype>
int isupper(int c);
isupper returns non-zero if c is an upper-case letter in the range A to Z, zero otherwise.

#include <cctype>
int isxdigit(int c);
isxdigit returns non-zero if c is a hexadecimal digit in the range a to f, A to F, or 0 to 9, zero otherwise.

#include <cstdlib>
long labs(long n);
labs returns as a long integer the absolute value of the long integer n.

#include <cmath>
double ldexp(double x, int n);
ldexp returns as a double floating-point number the result of x *(2**n).

#include <cstdlib>
ldiv_t ldiv(int n, int d);
ldiv calculates the quotient and remainder of n/d. The results are stored in the long members quot and rem of a structure of type ldiv_t. The type ldiv_t is defined in cstdlib.

#include <cmath>
double log(double x);
log returns as a double floating-point number the natural logarithm of x.

#include <cmath>
double log10(double x);
log10 returns as a double floating-point number the logarithm to base 10 of x.

#include <cstdlib>
void *malloc(size_t size);
malloc allocates space in memory for an object with size (in bytes) of size. The function returns a pointer to the allocated memory, or NULL if the memory could not be allocated. Memory allocated by malloc is not initialised to any particular value.

#include <cstring>
void *memchr(const void *s, unsigned char c, size_t n);
memchr returns a pointer to the first occurrence of the character c within the first n characters of the array s. The function returns NULL if there is no match. The type size_t is defined in stddef.h as an unsigned integer.

#include <cstring>
int memcmp(const void *s1, const void *s2, size_t n);
memcmp compares the first n characters of s1 with those of s2 and returns an integer less than, equal to or greater than zero depending on whether s1 is lexicographically less than, equal to or greater than s2.

#include <cstring>
void *memcpy(void *outs, const void *ins, size_t n);
memcpy causes n characters to be copied from the array ins to the array outs. The function returns a pointer to outs.

#include <cstring>
void *memmove(void *outs, const void *ins, size_t n);
memmove causes n characters to be copied from the array ins to the array outs, additionally allowing the copy to take place even if the objects being copied overlap in memory. The function returns a pointer to outs.

#include <cstring>
void *memset(void *s, unsigned char c, size_t n);
memset causes the first n characters of the array s to be filled with the character c. The function returns a pointer to s.

#include <cmath>
double modf(double x, double *iptr);
modf returns the fractional part of x and the integral part of x at the double pointer iptr.

#include <cstdio>
void perror(const char *s);
perror displays on the standard error device the string s, followed by a colon and an error message generated according to the contents of the value of the external variable errno which a corresponding declaration in errno.h.

#include <cmath>
double pow(double x, double y);
pow returns the value of x raised to the power of y as a double floating-point number.

#include <cstdio>
int printf(const char *<format>, ...);
printf writes to standard output the contents of the format string, other than special control sequences contained in the format string, followed by the contents of a list of variables converted according to the control sequences in the format string.

These are the printf format codes and their meanings:

d, i, o, u x, X The variable corresponding to the format code is converted to decimal (d,i), octal (o), unsigned decimal (u) or unsigned hexadecimal (x and X). The x conversion uses the letters abcdef; X uses ABCDEF.

f The variable is converted to a decimal notation of form [-]ddd.ddd, where the minimum width (w) of the field and the precision (p) are specified by %w.pf. The default precision is 6 characters; a precision of zero causes the decimal point to be suppressed.

e, E The float or double variable is converted to scientific notation of form [-]d.ddde±dd. Width and precision may also be specified. The default precision is 6 characters; a precision of zero causes the decimal point to be suppressed.

g, G The float or double variable is printed in style f or e. Style e is used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeroes are removed. A decimal point appears only if it is followed by a digit.

c The variable is displayed as a character.

s The variable is taken to be a string (character pointer) and characters from the string are displayed until a null character is encountered or the number of characters indicated by the precision specification is reached.

p Display variable as a pointer of type void *.

n The associated variable is a pointer to an integer which is assigned the number of characters displayed so far by printf on this call.

% Display a literal %.

A range of modifiers may be used with the format codes to specify the field width, signing and justification, precision and length of the converted output.

An integer between the percent sign and the format code specifies the minimum width of the output field. The output is padded, if necessary, with spaces, or with zeros if the integer is prefixed with a 0.

All output is, by default, right-justified; it can be left-justified by insertion of a - before the format code (and minimum width specifier, if any). Similar insertion of a + ensures the number is printed with a sign; a space character causes a space to prefix the output if there is no sign.

Precision is specified if the minimum width specifier is followed by a full-stop and an integer. The value of the integer specifies the maximum number of characters to be displayed from a string, or the minimum number of digits to be displayed for an integer, or the number of decimal places to be displayed, or the number of significant digits for output of floating-point data.

Length modifiers h, l and L are available. h causes the corresponding variable to be printed as a short; l as a long and L as a long double.

#include <cstdio>
int putc(int c, FILE *fp);
putc writes the character c to the file pointed to by fp and returns it; it returns EOF on error. putc is a macro and is equivalent to fputc.

This Page Intentionally Left Blank

Premier12